FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Modulare und parallele Implementierung des Jacobi-Davidson-Verfahrens

René Puttin

FZJ-ZAM-IB-2005-17

Dezember 2005

(letzte Änderung: 01.12.2005)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Die ersten vier Kapitel der Arbeit entstanden in Zusammenarbeit mit Britta Janssen, die zur gleichen Zeit ebenfalls eine Diplomarbeit zum Thema Jacobi-Davidson-Verfahren geschrieben hat. Für die gute Zusammenarbeit möchte ich mich an dieser Stelle recht herzlich bedanken.

Weiterhin möchte ich mich bei meinen Kollegen aus dem ZAM, die mich beim Umgang mit Programmbibliotheken und anderen Problemen unterstützt haben, bedanken.

Besonderer Dank gilt Herrn Willi Erkens und Herrn André Latour, die mich bei Detail-Fragen zur Programmiersprache Fortran beziehungsweise mathematischen Fragestellungen immer sehr gut beraten haben.

Zu guter Letzt möchte ich mich bei meiner Freundin Jessica Dück bedanken, dass sie während der ganzen Zeit so viel Zeit und Ruhe für mich hatte und mir ermöglicht hat mich voll auf die Arbeit zu konzentrieren.

Vielen Dank Ihnen beziehungsweise Euch Allen!

Zusammenfassung

In der Praxis treten an sehr unterschiedlichen Stellen Probleme auf, die auf die Berechnung von Eigenwerten hinauslaufen. Häufig handelt es sich dabei um Probleme sehr großer Dimensionen, die mangels Zeit und Speicherplatz nicht auf seriellen Rechnern gelöst werden können. Daher ist es erforderlich hoch-optimierte, parallele Programme zu verwenden, die innerhalb kurzer Zeit die gewünschten Eigenwerte großer Matrizen berechnen können.

Häufig ist es für Ingenieure auch überhaupt nicht notwendig alle Eigenwerte einer Matrix zu berechnen sondern nur die größten beziehungsweise kleinsten oder Eigenwerte um ein bestimmtes Ziel herum.

Das Jacobi-Davidson-Verfahren stellt genau diese Möglichkeiten zur Verfügung. Zur Berechnung der Eigenwerte wird das Ausgangsproblem in einen kleinen Unterraum projiziert. Mithilfe der leicht zu berechnenden Eigenpaare des projizierten Problems werden dann Näherungen für die Eigenpaare des ursprünglichen Problems berechnet.

Im Rahmen der Diplomarbeit wurde ein bestehendes paralleles Jacobi-Davidson-Programm auf den Supercomputer JUMP portiert und dort neu strukturiert. Weiterhin wurde es um ein Matrix-Format, die Möglichkeiten der Behandlung diagonalisierbarer unsymmetrischer Matrizen und der Berechnung innerer Eigenwerte erweitert.

Abstract

Many application problems require the calculation of eigenvalues. Often these are problems of huge dimensions, which cannot be solved on serial computers because of time and data space. Therefore it is necessary to use highly optimised parallel programs, which compute the desired eigenvalues of huge matrices in a short time.

Engineers often are not interested in all eigenvalues of a matrix, but in the largest or smallest or the eigenvalues near a specific value.

The Jacobi-Davidson-Method provides exactly these options. To compute the eigenvalues, the problem is projected into a small subspace. On the basis of the projected problems eigenvalues the algorithm computes estimations for the eigenpairs of the original problem.

In the context of this diploma thesis an existing parallel program was ported onto the supercomputer JUMP and reorganised there. Further on a new matrix format and the options of handling general diagonalizeable unsymmetric matrices and computing inner eigenvalues were implemented.

Kapitel 1

Einleitung

In der Diplomarbeit sollte ein bereits bestehendes Programm neu strukturiert und um einige Funktionalitäten erweitert werden. Das Programm berechnet mit Hilfe eines numerischen Verfahrens, des Jacobi-Davidson-Verfahrens, näherungsweise Eigenwerte und Eigenvektoren großer Matrizen.

Eigenwertprobleme treten in der Praxis an sehr unterschiedlichen Stellen auf. Eines der wichtigsten Einsatzgebiete ist die Mechanik. Hier beschreiben die Eigenwerte die Eigenschwingungen eines Bauteiles. Ingenieure benötigen die Eigenwerte um das Verhalten von Bauwerken, wie zum Beispiel Brücken, bei Erdbeben oder lang andauernder Windströmung einschätzen zu können.

Auch in anderen Bereichen spielen die Eigenwerte zur Beschreibung von Schwingungen eine wichtige Rolle. In der Quantenmechanik werden die Schwingungen der Atome anhand von Eigenwerten bestimmt. Im Bereich der Elektronik werden Schwingungen elektrischer Wellen, wie zum Beispiel Mikrowellen anhand von Eigenwerten berechnet.

Auch in der Stochastik spielen Eigenwerte eine große Rolle. Ein einfaches Beispiel sind die so genannten Markov-Ketten. Dies sind Prozesse, bei denen die Entscheidung zu einem bestimmten Zeitpunkt vollkommen unabhängig von der Vergangenheit ist. Zur Beschreibung einer Markov-Kette überprüft man häufig, ob diese eine bestimmte Anfangsverteilung, eine so genannte stationäre Startverteilung besitzt. Es lässt sich feststellen, dass die stationäre Startverteilung ein Eigenvektor zum Eigenwert eins der transponierten Übergangsmatrix ist. Die Markov-Ketten werden ausführlich in [5] diskutiert. Auch bei der Untersuchung komplexerer dynamischer Prozesse, wie zum Beispiel Wertpapieranalysen, spielen Eigenwerte häufig eine zentrale Rolle.

Da Eigenwerte in der Praxis so wichtig sind, werden dementsprechend auch sehr effiziente Programme zur Berechnung von Eigenwerten benötigt. Das Jacobi-Davidson-Programm, das in der Diplomarbeit weiter entwickelt wurde, soll die Grundlage für eine Bibliothek von hochoptimierten, parallelen Eigenwertlösern bilden.

Ausgangspunkt der Diplomarbeit war ein bestehendes Fortran77 Programm, das von Herrn Achim Basermann im Rahmen seiner Doktorarbeit [1] entwickelt wurde. Dieses parallele Programm wurde für ein Paragon-System implementiert und dient zur effizienten Berechnung von äußeren Eigenwerten symmetrischer dünnbesetzter Matrizen.

Im Rahmen der Diplomarbeit sollte das Programm auf den Superrechner JUMP portiert und dort optimiert und weiterentwickelt werden. Bei der Portierung des Programms musste die Speicherallokation neu implementiert werden, da diese zuvor mit Hilfe von C- und Paragon-System-Routinen durchgeführt wurde, die auf dem JUMP nicht zur Verfügung stehen.

Nach der Portierung stand die Reorganisation des Programms an. Ausgangspunkt für diese ist, dass das Jacobi-Davidson-Verfahren die System-Matrix nur in Form von Matrix-Vektor-Produkten verwendet. Dadurch ist es möglich die Matrix-Verwaltung vollständig vom Hauptalgorithmus abzukoppeln und in ein Modul auszulagern. In ähnlicher Art und Weise wurde ein Modul zur Lösung einer Korrekturgleichung implementiert.

Zum Test der Reorganisation wurde ein Format zur Speicherung von Band-Matrizen implementiert. Weiterhin wurde für dieses Format ein spezieller Bandlöser implementiert, mit dem auch das Modul zur Lösung der Korrekturgleichung weiter getestet werden konnte.

Das Programm wurde darüber hinaus um die Möglichkeit erweitert innere Eigenwerte und Eigenwerte symmetrisierbarer Matrizen zu berechnen.

Die Diplomarbeit ist wie folgt aufgebaut: Im zweiten Kapitel werden die grundlegenden Begriffe der linearen Algebra erläutert. Das dritte Kapitel stellt kurz einige grundlegende numerische Algorithmen vor, die später direkt oder zum Vergleich verwendet werden. Im vierten Kapitel wird die Theorie des Jacobi-Davidson-Verfahrens beschrieben. Das fünfte Kapitel beschreibt die, im Programm verwendeten, Formate zur Speicherung der Systemmatrizen. Im sechsten Kapitel werden die Schnittstellen-Module zur Verwaltung der Matrizen und Lösung der Korrekturgleichung beschrieben. Das siebte Kapitel beschreibt das entwickelte Programm und die einzelnen neu entwickelten Routinen. Das achte Kapitel stellt eine Benutzeranleitung dar. Im neunten Kapitel werden die Testergebnisse vorgestellt.

Kapitel 2

Mathematische Grundlagen

2.1 Matrizen und Vektoren

Nachdem in der Einleitung die praktische Bedeutung der Eigenwerte erklärt wurde, werden nun im zweiten Kapitel die entsprechenden mathematischen Definitionen gegeben. Dazu wird eine kurze Zusammenfassung der, im Hinblick auf die Arbeit, wichtigen Begriffe der Linearen Algebra gegeben. Eine ausführliche Beschreibung der Begriffe wird in [11] geliefert. Alle Aussagen, die fürs Reelle getroffen werden, gelten, falls dies nichts anderes erwähnt wird, analog auch im Komplexen. Aufbauend auf diesen Definitionen werden dann in den nächsten Kapiteln die numerischen Verfahren zur Bestimmung von Eigenwerte erläutert.

Wie im ersten Kapitel dargestellt, laufen viele Probleme auf lineare Gleichungssysteme

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & \vdots & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array}$$

hinaus. Diese lassen sich einfacher in Form von Matrizen und Vektoren, die im Folgenden vorgestellt werden, darstellen.

Bezeichnung 1 *Sei x ein m-Tupel von reellen Zahlen*

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

dann bezeichnet man x als m-dimensionalen Vektor. Man schreibt auch: $x \in \mathbb{R}^m$.

Bezeichnung 2 *Sei A ein zweidimensionales Anordnungssystem bestehend aus Zeilen und Spalten von reellen Werten*

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

dann bezeichnet man A als $m \times n$-dimensionale Matrix. Man schreibt auch: $A \in \mathbb{R}^{m \times n}$.

Analog zu den Skalaren lassen sich auch für Matrizen und Vektoren Addition und Multiplikation definieren.

Bemerkung 1 *Die Addition von Vektoren und Matrizen erfolgt komponentenweise.*

Eine Multiplikation von Vektoren ist durch das Skalarprodukt gegeben.

Definition 1 *Seien $x, y \in \mathbb{R}^n$ Vektoren, dann nennt man*

$$< x, y >= \sum_{i=1}^{n} x_i y_i$$

Skalarprodukt der Vektoren x und y.

Im Komplexen beinhaltet das Skalarprodukt die komplex konjugierten Komponenten des ersten Vektors.

Bezeichnung 3 *Seien $x, y \in \mathbb{R}^n$ und $z = x + iy$, dann nennt man*

$$\overline{z} = x - iy$$

die komplex konjugierte Zahl zur komplexen Zahl z.

Definition 2 *Seien $u, v \in \mathbb{C}$ komplexe Vektoren, dann heißt*

$$< u, v >= \sum_{i=1}^{n} \overline{u}_i v_i$$

Skalarprodukt der Vektoren u und v.

Bemerkung 2 *Gilt für das Skalarprodukt zweier Vektoren x und y*

$$< x, y >= 0$$

so stehen die beiden Vektoren senkrecht aufeinander. Man sagt auch, die Vektoren sind orthogonal und schreibt: $x \perp y$.

Um die Länge von Vektoren messen zu können, führt man so genannte Normen ein.

Definition 3 *Sei $x \in \mathbb{R}^n$ ein Vektor, dann heißt*

$$\|x\|_p = \left\{ \sum_{i=1}^{n} x_i^p \right\}^{1/p}$$

für $1 \leq p < \infty$ p-Norm des Vektors x.

Bemerkung 3 *Die 2-Norm, auch euklidische Norm genannt,*

$$\|x\|_2 = \sqrt{< x, x >}$$

beschreibt die Länge des Vektors x.

Häufig wird für die folgenden Definitionen die transponierte beziehungsweise komplex konjugiert transponierte Matrix benötigt.

Bezeichnung 4 *Sei $A \in \mathbb{R}^{m \times n}$ eine Matrix, dann nennt man*

$$A^T \text{ mit } a_{ij}^T = a_{ji} \quad \forall i = 1, ..., n, j = 1, ..., m$$

die transponierte Matrix zur Matrix A.

Bezeichnung 5 *Sei $A \in \mathbb{C}^{m \times n}$ eine Matrix, dann nennt man*

$$A^* \text{ mit } a_{ij}^* = \overline{a}_{ji} \quad \forall i = 1, ..., m, j = 1, ..., n$$

die komplex konjugiert transponierte Matrix zur Matrix A.

Damit lässt sich auch für Matrizen ein Produkt definieren.

Definition 4 *Seien $A \in \mathbb{R}^{m\times n}$ und $B \in \mathbb{R}^{n\times k}$ Matrizen, dann nennt man*

$$AB = C \text{ mit } c_{ij} =< (A^T)^{(i)}, B^{(j)} > \quad \forall i = 1, ..., m, j = 1, ..., k$$

Matrix-Matrix-Multiplikation der Matrizen A und B. Dabei steht $A^{(i)}$ für den i-ten Spaltenvektor der Matrix A.

Bemerkung 4 *In gleicher Weise lässt sich auch die Multiplikation einer $m \times n$-Matrix mit einem n-dimensionalen Vektor beschreiben. Der Vektor wird dabei als $n \times 1$-Matrix aufgefasst.*

$$Ax = y \text{ mit } y_i =< (A^T)^{(i)}, x > \quad \forall i = 1, ..., n$$

2.2 Lineare Gleichungssysteme

Nun ist man in der Lage, ein lineares Gleichungssystem in einer einfacheren Form aufzuschreiben, die in komplexeren Rechnungen deutlich leichter zu handhaben ist.

Bezeichnung 6 *Seien $A \in \mathbb{R}^{m\times n}$ eine Matrix und $x \in \mathbb{R}^n, b \in \mathbb{R}^m$ Vektoren, dann nennt man*

$$Ax = b$$

ein lineares Gleichungssystem.

Auch für Matrizen lassen sich inverse und neutrale Elemente definieren.

Bemerkung 5 *Die Nullmatrix*

$$O \in \mathbb{R}^{m\times n} \text{ mit } o_{ij} = 0 \quad \forall i = 1, ..., m, j = 1, ..., n$$

stellt das neutrale Element bezüglich der Matrizenaddition dar.

Bemerkung 6 *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, dann stellt die Matrix $-A$ das inverse Element von A bezüglich der Addition dar.*

Bemerkung 7 *Die Einheitsmatrix*

$$E \in \mathbb{R}^{n\times n} \text{ mit } e_{ij} = \delta_{ij} = \begin{cases} 1 \text{ für } i = j \\ 0 \text{ sonst} \end{cases}$$

stellt das neutrale Element bezüglich der Matrizenmultiplikation dar. Häufig wird die Einheitsmatrix auch mit I bezeichnet.

Bemerkung 8 *Sei $A \in \mathbb{R}^{n\times n}$, dann stellt*

$$A^{-1} \text{ mit } AA^{-1} = E$$

das inverse Element der Matrix A bezüglich der Multiplikation dar. Im Allgemeinen nennt man A^{-1} die Inverse von A.

Bemerkung 9 *Es ist zu beachten, dass die Inverse nur für quadratische Matrizen (siehe Bezeichnung 10) definiert ist und selbst dann nicht immer existiert.*

Für nichtquadratische Matrizen (siehe Bezeichnung 10), lässt sich eine Art Näherungsinverse definieren.

Bemerkung 10 *Sei $A \in \mathbb{R}^{m\times n}$, dann nennt man eine Matrix $A^+ \in \mathbb{R}^{n\times m}$ mit den Eigenschaften*

- $A = AA^+A$
- $A^+ = A^+AA^+$

Pseudoinverse zur Matrix A

Im Folgenden soll nun genauer betrachtet werden, wann eine inverse Matrix existiert.

Definition 5 *Seien $x_i \in \mathbb{R}^n, i = 1, ..., m$ Vektoren und $\alpha_i \in \mathbb{R}, i = 1, ..., m$ reelle Zahlen, dann heißen die x_i linear unabhängig, falls gilt:*

$$\sum_{i=1}^{m} \alpha_i x_i = 0 \Leftrightarrow \alpha_i = 0 \quad \forall i = 1, ..., m$$

Bezeichnung 7 *Die Anzahl der linear unabhängigen Spaltenvektoren einer Matrix bezeichnet man als Spaltenrang. Analog dazu bezeichnet man die Anzahl der linear unabhängigen Zeilenvektoren als Zeilenrang. Es lässt sich feststellen, dass für jede Matrix der Spalten- und Zeilenrang identisch sind. Deshalb spricht man in der Regel nur vom Rang einer Matrix.*

Bezeichnung 8 *Sei $A \in \mathbb{R}^{n \times n}$ eine Matrix, dann heißt A invertierbar, falls gilt:*

$$rang(A) = n$$

Bemerkung 11 *Seien $A \in \mathbb{R}^{n \times n}$ eine Matrix und $x, b \in \mathbb{R}^n$ Vektoren, dann hat das Gleichungssystem*

$$Ax = b$$

genau dann eine eindeutige Lösung, wenn A invertierbar ist. Als eindeutige Lösung ergibt sich:

$$x = A^{-1}b$$

Löst man ein Gleichungssystem numerisch, so benötigt man eine Aussage darüber, wie genau die aktuelle Näherung an der exakten Lösung liegt. Diese liefert das Residuum.

Bezeichnung 9 *Seien $A \in \mathbb{R}^{n \times n}$ eine Matrix und $x, \hat{x}, b \in \mathbb{R}^n$ Vektoren, wobei $\hat{x}$ eine Näherungslösung des Gleichungssystems*

$$Ax = b$$

darstellt, dann nennt man

$$r = b - A\hat{x}$$

das Residuum der Näherung.

2.3 Arten von Matrizen

Matrizen lassen sich aufgrund ihrer Eigenschaften in verschiedene Klassen einteilen.

Bezeichnung 10 *Sei $A \in \mathbb{R}^{m \times n}$ eine Matrix, dann heißt A quadratisch, falls gilt:*

$$m = n$$

Ansonsten heißt A nichtquadratisch.

Bezeichnung 11 *Sei $A \in \mathbb{R}^{n \times n}$ eine Matrix, dann heißt A symmetrisch falls gilt:*

$$A = A^T$$

Ansonsten heißt A unsymmetrisch.

Bezeichnung 12 *Sei $A \in \mathbb{R}^{n \times n}$ eine unsymmetrische Matrix, dann heißt A symmetrisierbar falls A nur reelle Eigenwerte (siehe Definition 6) besitzt.*

Bezeichnung 13 *Sei $A \in \mathbb{C}^{n \times n}$ eine Matrix, dann heißt A hermitesch, falls gilt:*

$$A = A^*$$

Bezeichnung 14 *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A anti- oder schiefsymmetrisch, falls gilt:*

$$A = -A^T$$

Bezeichnung 15 *Sei $A \in \mathbb{C}^{n\times n}$ eine Matrix, dann heißt A anti- oder schiefhermitesch, falls gilt:*

$$A = -A^*$$

Bezeichnung 16 *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix, dann heißt A positiv definit, genau dann, wenn gilt:*

$$< x, Ax > > 0 \quad \forall x \in \mathbb{R}^n$$

Bezeichnung 17 *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, dann heißt A diagonal dominant, falls gilt:*

$$a_{ii} \geq \sum_{i\neq j} a_{ij} \quad \forall i = 1, ..., m$$

Sind die Diagonalelemente echt größer als die Summe, so nennt man die Matrix streng diagonal dominant. Falls diese wesentlich größer sind, wird die Matrix als extrem diagonal dominant bezeichnet.

Bezeichnung 18 *Sei $Q \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt Q orthogonal, falls gilt:*

$$QQ^T = E \Leftrightarrow Q^T = Q^{-1}$$

Bezeichnung 19 *Sei $U \in \mathbb{C}^{n\times n}$ eine Matrix, dann heißt U unitär, falls gilt:*

$$UU^* = E \Leftrightarrow U^* = U^{-1}$$

Bemerkung 12 *Die Spaltenvektoren $q^{(i)}$ einer orthogonalen Matrix Q bilden ein Orthonormalsystem, das heißt:*

$$\|q^{(i)}\|_2 = 1 \quad \forall i = 1, ..., n$$

$$< q^{(i)}, q^{(j)} >= \delta_{ij} = \begin{cases} 1 \text{ für } i = j \\ 0 \text{ sonst} \end{cases} \qquad \forall i, j = 1, ..., n$$

Neben ihren Eigenschaften unterscheidet man Matrizen anhand ihrer Besetzungsstrukturen.

Bezeichnung 20 *Sei $A \in \mathbb{R}^{n\times m}$ eine Matrix, dann heißt A untere beziehungsweise obere Dreiecksmatrix, falls gilt:*

$$a_{ij} = 0 \quad \forall i > j \text{ bzw. } a_{ij} = 0 \quad \forall i < j$$

Bemerkung 13 *Eine obere / untere Dreiecksmatrix mit genau einer zusätzlichen Diagonale unterhalb / oberhalb der Hauptdiagonalen bezeichnet man als Hessenbergmatrix. Man sagt auch: Die Matrix befindet sich in Hessenberg-Form.*

Bezeichnung 21 *Sei $A \in \mathbb{R}^{n\times m}$ eine Matrix, dann heißt A Bandmatrix mit n_l Sub- (Nebendiagonalen links / unterhalb der Hauptdiagonalen) und n_r Superdiagonalen (Nebendiagonalen rechts / oberhalb der Hauptdiagonalen), falls gilt:*

$$a_{ij} = 0 \quad \forall a_{ij} \text{ mit } i - j > n_l \vee j - i > n_r$$

Bemerkung 14 *Bandmatrizen mit jeweils einer Sub- und Superdiagonalen bezeichnet man als Tridiagonalmatrizen.*

Bezeichnung 22 *Neben den klassischen Besetzungsarten existieren auch willkürlich dünn besetzte Matrizen, die man als Sparse Matrizen bezeichnet.*

Eine Matrix, die später in der Arbeit häufiger benötigt wird, ist die so genannte charakteristische Matrix.

Bezeichnung 23 *Seien $A \in \mathbb{R}^{n\times n}$ eine quadratische Matrix und λ einer ihrer Eigenwerte (siehe Definition 6), dann nennt man die Matrix $A - \lambda I$ charakteristische Matrix.*

2.4 Eigen- und Ritzwerte

In der Numerik werden Matrizen in zwei grundlegenden Problemstellungen behandelt: Das erste Problem ist die Lösung von Gleichungssystemen, welche bereits im Kapitel 2.2 vorgestellt wurde. Das zweite Problem ist das so genannte Eigenwertproblem.

Definition 6 *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix, $x \in \mathbb{R}^n, x \neq 0$ ein Vektor und λ ein Skalar, die die Gleichung*

$$Ax = \lambda x$$

erfüllen, dann nennt man λ Eigenwert und x Eigenvektor der Matrix A. (λ, x) wird auch als Eigenpaar bezeichnet.

Bemerkung 15 *Man unterscheidet die Eigenwerte in äußere und innere Eigenwerte. Die äußeren Eigenwerte sind die größten beziehungsweise kleinsten Eigenwerte einer Matrix. Diese können im Allgemeinen leichter berechnet werden. Die inneren Eigenwerte sind die Eigenwerte im inneren des Eigenwert-Intervalls.*

Bemerkung 16 *Die Eigenwerte von A entsprechen den Nullstellen des charakteristischen Polynoms*

$$\det(A - \lambda I) = 0$$

Es lässt sich direkt erkennen, dass für Eigenwertprobleme mit einer Matrixdimension größer als vier mit Hilfe des charakteristischen Polynoms keine analytische Lösung mehr gefunden werden kann, da für Gleichungen mit höherer Ordnung als vier keine geschlossene Lösungsformel existiert.

Auch für das Eigenwertproblem lässt sich ein Residuum definieren, mit dem man in numerischen Rechnungen Fehlerschranken aufstellen kann.

Bezeichnung 24 *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix und $\theta \in \mathbb{R}$ und $y \in \mathbb{R}^n, y \neq 0$ Näherungen für einen Eigenwert und den zugehörigen Eigenvektor der Matrix A. Dann bezeichnet man mit*

$$r = Ay - \theta y$$

das Residuum des genäherten Eigenpaares.

Bei der numerischen Bestimmung der Eigenwerte großer Matrizen werden häufig Unterraum-Techniken verwendet.

Definition 7 *$U \subset \mathbb{R}^n$ heißt Unterraum des $\mathbb{R}^n$, falls gilt:*

- $U \neq \emptyset$
- $x + y \in U \quad \forall x, y \in U$ *(Additivität)*
- $\alpha \cdot x \in U \quad \forall x \in U, \alpha \in \mathbb{R}$ *(Homogenität)*

Ein Unterraum wird durch seine Basis festgelegt.

Definition 8 *Seien $U \subset \mathbb{R}^n$ ein Unterraum und $u_i \in \mathbb{R}^n, i = 1, ..., k$ Vektoren, dann heißt $\{u_1, ..., u_k\}$ Basis des Unterraums U, falls*

- *die Vektoren u_i linear unabhängig sind*
- *die lineare Hülle $L = \left\{\sum_{i=1}^k \alpha_i u_i, \alpha_i \in \mathbb{R}, i = 1, ..., n\right\} = U$ ist*

Weiterhin bezeichnet man mit $dim(U) = k$ die Dimension des Unterraums.

Die meisten Unterraum-Verfahren arbeiten mit speziellen Unterräumen, so genannten Krylov-Unterräumen.

Definition 9 *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix und $q_0 \in \mathbb{R}$ ein gegebener Vektor, dann bezeichnet man mit*

$$K_{k+1}(A, q_0) = span(q_0, Aq_0, ..., A^k q_0)$$

den von A über q_0 erzeugten Krylov-Raum.

Für Unterräume lassen sich analog zu den Eigenwerten und Eigenvektoren die Ritzwerte und Ritzvektoren definieren.

Definition 10 *Seien U ein Unterraum des $\mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$ eine Matrix, $z \in U, z \neq 0$ ein Vektor und θ ein Skalar, die die Ritz-Galerkin Bedingung*

$$Az - \theta z \perp U$$

erfüllen, dann nennt man θ Ritzwert und z Ritzvektor zu A bezüglich des Unterraums U. (θ, z) wird auch als Ritzpaar bezeichnet.

Kapitel 3

Grundlegende Algorithmen

In diesem Kapitel werden kurz einige grundlegende mathematische Verfahren beschrieben, die später in den komplexeren Eigenwert-Verfahren verwendet werden.

3.1 Gram-Schmidt-Verfahren

Die Krylov-Unterraum-Verfahren, die im nächsten Kapitel vorgestellt werden, arbeiten häufig mit orthonormalen Basen für die Unterräume. Die Unterräume werden dabei sukzessiv erweitert, das heißt in jedem Iterationschritt wird zu einer bestehenden Basis ein neuer Basisvektor hinzugefügt. Das Orthogonalisierungsverfahren von Gram-Schmidt liefert dazu einen passenden Algorithmus, der einen Vektor t bezüglich der orthonormalen Vektoren $v_i, i = 1, ..., m-1$ orthonormalisiert.

for $i = 1, .., m-1$
  $\alpha_i =< v_i, t >$
end for
for $i = 1, .., m-1$
  $t = t - \alpha_i v_i$
end for
$t = \frac{t}{\|t\|_2}$

Abbildung 3.1: Standard Gram-Schmidt-Verfahren

Da das Standard Gram-Schmidt-Verfahren numerisch instabil ist, verwendet man häufig das modifizierte Gram-Schmidt-Verfahren.

for $i = 1, .., m-1$
  $t = t- < v_i, t > v_i$
end for
$t = \frac{t}{\|t\|_2}$

Abbildung 3.2: Modifiziertes Gram-Schmidt-Verfahren

3.2 Arnoldi- und Lanczos-Verfahren

Um das Jacobi-Davidson-Verfahren später besser im Rahmen der Eigenwert-Verfahren einordnen zu können, werden im Folgenden kurz zwei grundlegende Eigenwertwert-Löser, die Arnoldi und die Lanczos-Methode vorgestellt. Eine ausführliche Herleitung der Verfahren findet man in [13].

Der Arnoldi-Algorithmus dient als Projektionsmethode für $A \in \mathbb{R}^{n \times n}$, n groß. Er liefert Orthonormalbasen von Krylov-Räumen. Den Krylov-Raum benutzt man dann als Suchraum für die Projektionsmethode. Der Arnoldi-Algorithmus berechnet zu einer Matrix A eine unitäre Matrix $Q = [q_1, ..., q_n]$, so dass $Q^{(-1)}AQ = H$ die Hessenbergform besitzt. H stellt die Galerkin-Projektion von A in den Krylov-Raum dar. Berechnet man nun die Eigenpaare von H, so sind dies Ritzpaare von A bezüglich des aufgestellten Krylov-Raums und somit Näherungen für ein Eigenpaar von A.

$q_1 = \frac{r_0}{\|r_0\|_2}$
for $j = 1, ..., k-1$
  $w_{j+1} = Aq_j$
  $\tilde{w}_{j+1} = w_{j+1} - \sum_{i=1}^{j} q_i < q_i, w_{j+1} >$
  if $\tilde{w}_{j+1} = 0$ **then**
    stop
  else
    $q_{j+1} = \frac{\tilde{w}_{j+1}}{\|\tilde{w}_{j+1}\|_2}$
  end if
end for

Abbildung 3.3: Arnoldi-Verfahren

Für den Spezialfall, dass $A = A^*$ (A hermitesch), entspricht der Lanczos-Algorithmus im Wesentlichen dem Arnoldi-Algorithmus. In diesem Fall ist die Hessenbergmatrix H tridiagonal und es kann eine 3-Term-Rekursion angewendet werden, das heißt es müssen nicht alle alten q_i gespeichert werden.

wähle $x \neq 0$
$v_0 = 0$
$\beta_0 = 0$
$v_1 = \frac{x}{\|x\|_2}$
for $j = 1, ..., k$
  $\alpha_j = v_j^* A v_j$
  $r_j = Av_j - \beta_{j-1}v_{j-1} - \alpha_j v_j$
  if $r_j = 0$ **then**
    stop
  else
    $\beta_j = \|r_j\|_2$
    $v_{j+1} = \frac{1}{\beta_j} r_j$
  end if
end for

Abbildung 3.4: Lanczos-Verfahren

Sowohl beim Lancos als auch bei Arnoldi-Verfahren treten die vorzeitigen Abbrüche sehr selten auf, da es sehr selten vorkommt, dass nach längerer Rechnung ein Ergebnis exakt 0 ist.

3.3 Konjugierte Gradienten-Verfahren

Symmetrische lineare Gleichungssyteme können mit dem klassischen CG-Verfahren (konjugierte Gradienten = **C**onjugated **G**radients) gelöst werden.

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig
$g^{(0)} = b - Ax^{(0)}, d^{(0)} = g^{(0)}$
$k = 0$
while $\|g^{(k)}\| > \varepsilon$
$\alpha_k = \frac{g^{(k)^T} g^{(k)}}{d^{(k)^T} A d^{(k)}}$
$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
$g^{(k+1)} = g^k + \alpha_k A d^{(k)}$
$\beta_k = \frac{g^{(k+1)^T} g^{(k+1)}}{g^{(k)^T} g^{(k)}}$
$d^{k+1} = -g^{(k+1)} + \beta_k d^{(k)}$
$k = k + 1$
end while

Abbildung 3.5: Klassisches Konjugierte-Gradienten Verfahren

Sind die Matrizen indefinit oder unsymmetrisch, so konvergiert das klassische CG-Verfahren in der Regel nicht und man muss es modifizieren. CG-artige Verfahren, die auch dann konvergieren sind das QMR- (**Q**uasi **M**inimal **R**esidual) und das TFQMR-Verfahren (**T**ranspose **F**ree **Q**uasi **M**inimal **R**esidual). Eine ausführliche Beschreibung der beiden Verfahren ist in [1] zu finden. In [12] werden weitere CG-artige Verfahren vorgestellt. Das QMR-Verfahren kombiniert die Lanczos-Methode mit dem Ansatz der quasi-minimalen Residuen. Allerdings wird in jedem Schritt nicht die Norm der Residuen, sondern jeweils nur ein Faktor minimiert, wodurch sich der Aufwand des zu lösenden Minimierungsproblems stark verringert. Insgesamt ergibt sich daraus mit dem Vorkonditionierer $M = M_1 M_2$ der folgende Algorithmus:

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$g^{(0)} = Ax^{(0)} - b, \tilde{g}^{(1)} = -g^{(0)}$

löse $M_1 y = \tilde{g}^{(1)}$

$\rho_1 = \|y\|_2$

wähle $\tilde{w}^{(1)}$ beliebig, zum Beispiel: $\tilde{w}^{(1)} = -g^{(0)}$

löse $M_2^T t = \tilde{w}^{(1)}$

$\xi_1 = \|t\|_2, \gamma_0 = 1, \eta_0 = -1$

while $\|g^{(i)}\|_2 \leq \epsilon_r$

$q^{(i)} = \frac{\tilde{q}^{(i)}}{\rho_i}, y = \frac{y}{\rho_i}, w^{(i)} = \frac{\tilde{w}^{(i)}}{\xi_i}, t = \frac{t}{\xi_i}, \delta_i = t^T y$

löse $M_2 \tilde{y} = y$, löse $M_1^T \tilde{t} = t$

if $i = 1$ **then**

$p^{(1)} = \tilde{y}, v^{(1)} = \tilde{t}$

else

$p^{(1)} = \tilde{y} - \frac{\xi_i \delta_i}{\varepsilon_{i-1}} p^{(i-1)}, v^{(1)} = \tilde{t} - \frac{\rho_i \delta_i}{\varepsilon_{i-1}} v^{(i-1)}$

end if

$\varepsilon_i = {v^{(i)}}^T A p^{(i)}, \beta_i = \frac{\varepsilon_i}{\delta_i}, \tilde{q}^{(i+1)} = Ap^{(i)} - \beta_i q^{(i)}$

löse $M_1 y = \tilde{q}^{(i+1)}$

$\rho_{i+1} = \|y\|_2, \tilde{w}^{(i+1)} = A^T v^{(i)} - \beta_i w^{(i)}$

löse $M_2^T t = \tilde{w}^{(i+1)}$

$\xi_{i+1} = \|t\|_2, \vartheta_i = \frac{\rho_{i+1}}{\gamma_{i-1}|\beta_i|}, \gamma_i = \frac{1}{\sqrt{1+\vartheta_i^2}}, \eta_i = \frac{-\eta_{i-1}\rho_i \gamma_i^2}{\beta_i \gamma_{i-1}^2}$

if $i = 1$ **then**

$d^{(i)} = \eta_i p^{(i)}, s^{(i)} = \eta_i A p^{(i)}$

else

$d^{(i)} = \eta_i p^{(i)} + (\vartheta_{i-1}\gamma_i)^2 d^{(i-1)}, s^{(i)} = \eta_i A p^{(i)} + (\vartheta_{i-1}\gamma_i)^2 s^{(i-1)}$

end if

$x^{(i)} = x^{(i-1)} + d^{(i)}, g^{(i)} = g^{(i-1)} + s^{(i)}$

end while

Abbildung 3.6: Quasi Minimal Residual-Verfahren

Das QMR-Verfahren verwendet an einigen Stellen Matrix-Vektor-Produkte mit transponierten Matrizen. Das Transponieren ist aber, insbesondere auf Parallelrechnern, eine sehr aufwändige Operation. Diese wird durch das transpositionsfreie TFQMR-Verfahren vermieden. Der Algorithmus zur TFQMR-Methode sieht wie folgt aus:

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$g^{(0)} = Ax^{(0)} - b, y^{(1)} = -g^{(0)}, w^{(1)} = -g^{(0)}, v^{(0)} = Ay^{(1)}$

$d^{(0)} = 0, \tau_0 = \|g^{(0)}\|_2, \vartheta_0 = 0, \eta_0 = 0$

wähle $\tilde{g}^{(0)}$ so, dass $\rho_0 = -\tilde{g}^{(0)^T} g^{(0)} \neq 0$

while $x^{(2i-1)}$ und $x^{(2i)}$ nicht konvergiert

$\sigma_{i-1} = \tilde{g}^{(0)^T} v^{(i-1)}, \alpha_{i-1} = \frac{\rho_{i-1}}{\sigma_{i-1}}, y^{(2i)} = y^{(2i-1)} - \alpha_{i-1} v^{(i-1)}$

for $j = 2i-1, 2i$

$w^{(j+1)} = w^{(j)} - \alpha_{i-1} Ay^{(j)}, \vartheta_j = \frac{\|w^{(j+1)}\|_2}{\tau_{j-1}}$

$\gamma_j = \frac{1}{\sqrt{1+\theta_j^2}}, \tau_j = \tau_{j-1} \vartheta_j \gamma_j, \eta_j = \gamma_j^2 \alpha_{i-1}$

$d^{(j)} = y^{(j)} + \frac{\vartheta_{j-1}^2 \eta_{j-1}}{\alpha_{i-1}} d^{(j-1)}, x^{(j)} = x^{(j-1)} + \eta_j d^{(j)}$

end for

$\rho_i = \tilde{g}^{(0)^T} w^{(2i+1)}, \beta_i = \frac{\rho_i}{\rho_{i-1}}, y^{(2i+1)} = w^{(2i+1)} \beta_i y^{(2i)}$

$v^{(i)} = Ay^{(2i+1)} + \beta_i (Ay^{(2i)} + \beta_i v^{(i-1)})$

end while

Abbildung 3.7: Transpose Free Quasi Minimal Residual-Verfahren

Kapitel 4

Das Jacobi-Davidson-Verfahren

Im Rahmen der Diplomarbeit sollte ein bestehendes Jacobi-Davidson Programm reorganisiert und weiter entwickelt werden. In diesem Kapitel wird nun das Jacobi-Davidson-Verfahren erläutert. Um dieses leichter zu verstehen, werden zunächst die beiden zu Grunde liegenden Verfahren, das Davidson und das Jacobi-Verfahren, vorgestellt. Die Grundlage für diese Kapitel liefert die Herleitung des Jacobi-Davidson-Verfahrens in [17].

4.1 Das Davidson-Verfahren

Die Hauptidee des Davidson-Verfahrens ist folgende: Man geht von einem Unterraum κ der Dimension k mit einer orthonormalen Basis $u_1, ..., u_k$ aus. Sei nun U_k die Matrix mit den Spalten $u_1, ..., u_k$, dann lässt sich die Matrix A reduziert auf den Unterraum schreiben als $B_k = U_k^* A U_k$. Sei nun (θ, s) ein Eigenpaar von B_k, dann gilt:

$$B_k s = \theta s$$

Im Hinblick auf den Unterraum sind die Eigenwerte der Matrix B_k die Ritzwerte der Matrix A und $y = U_k s$ liefert den Ritzvektor von A zu einem Eigenvektor s der Matrix B_k, denn θ und $U_k s$ erfüllen die Ritz-Galerkin Bedingung für das Residuum des angenäherten Eigenpaares:

$$\begin{aligned} & r = AU_k s - \theta U_k s \perp \{u_1, ..., u_k\} \\ \Leftrightarrow \quad & U_k^* r = U_k^* A U_k s - \theta s = B_k s - \theta s = 0 \end{aligned} \tag{4.1}$$

Daraus lässt sich weiterhin ableiten:

$$\begin{aligned} r &= AU_k s - \theta U_k s = Ay - \theta y \\ \Leftrightarrow y &= (A - \theta I)^{-1} r \end{aligned}$$

Davidson arbeitete als Chemiker hauptsächlich mit streng diagonal dominanten Matrizen. Bei diesen kann man den „Shift-and-Invert"-Schritt $(A - \theta I)^{-1}$ annähern durch $(D_A - \theta I)^{-1}$, wobei D_A eine Diagonalmatrix mit den Diagonalelementen von A darstellt. Dadurch erhält man einen neuen Vektor

$$t = (D_A - \theta I)^{-1} r$$

der zur Erweiterung des Unterraums verwendet wird. Der Vektor t wird bezüglich des Unterraums U_k orthonormalisiert und kann somit zur bestehenden Basis hinzugenommen werden.

Verwendet man an dieser Stelle keine Näherung der Matrix A sondern rechnet mit der Matrix A selbst, so erhält man wiederum y. Diese Richtung ist aber bereits in der Basis enthalten, also wird sie nicht erweitert. Auch für extrem diagonal dominante Matrizen stagniert das Verfahren. Dies lässt

sich leicht erkennen, indem man von einer Diagonalmatrix A ausgeht. Die Matrix $D_A - \theta I$ enthält in diesem Fall mindestens eine Nullzeile und ist somit nicht invertierbar.

Ursprünglich wurde das Davidson-Verfahren für symmetrische Matrizen entwickelt. Es hat sich aber gezeigt, dass es ebenso für unsymmetrische Matrizen geeignet ist. In Abbildung 4.1 ist das Davidson Verfahren für unsymmetrische Matrizen im Detail dargestellt. Bei symmetrischen Matrizen müssen nur ungefähr die Hälfte an inneren Produkten bei der Konstruktion der B_k berechnet werden.

$u_1 = \frac{v}{\|v\|_2}$
for $k = 1, ..., m$
  $w_k = Au_k$
  berechne die k-te Zeile und Spalte von B_k
  for $j = 1, ..., k-1$
    $B(j,k) = u_j^* w_k$
    $B(k,j) = u_k^* w_j$
  end for
  $B(k,k) = u_k^* w_k$
  berechne den größten Eigenwert θ von B_k und den zugeh. Eigenvektor s
  $z = \sum_{j=1}^{k} s_{(j)} u_j$
  $r = Az - \theta z$
  $t = (D_A - \theta I)^{-1} r$
  for $j = 1, ..., k$
    $t = t - < u_j^* t > u_j$
  end for
  $u_{k+1} = \frac{t}{\|t\|_2}$
end for

Abbildung 4.1: Davidson-Verfahren mit Diagonal-Vorkonditionierer

4.2 Das Verfahren von Jacobi

Im Verfahren von Jacobi werden zwei Schritte durchgeführt um Eigenwerte zu bestimmen. Im ersten Schritt wird das klassische Jacobi-Verfahren zur Eigenwertberechung symmetrischer Matrizen angewendet und die Matrix A, von der die Eigenwerte bestimmt werden sollen, wird durch Rotationen auf die Form

$$D_k \cdot ... \cdot D_1 \cdot A \cdot D_{-1} \cdot ... \cdot D_{-k} = \begin{pmatrix} \alpha & c^T \\ b & F \end{pmatrix}$$

gebracht. Von dem erzeugten System ausgehend wird dann im zweiten Schritt ein lineares System für das orthogonale Komplement der genäherten Eigenvektoren abgeleitet. Dieses wird dann mit Gauß-Jacobi Iterationen gelöst. Die Konstruktion des linearen Systems wird auch als „Jacobi's orthogonal complement correction“ (JOCC) bezeichnet, welche im Folgenden erklärt wird. Diese Methode war lange Zeit in Vergessenheit geraten und wurde von Van der Vorst [17] und Sleijpen im Jacobi-Davidson-Verfahren wieder neu aufgegeriffen.

Seien nun A eine streng diagonal dominante Matrix und ohne Beschränkung der Allgemeinheit $a_{1,1} = \alpha$ das größte Diagonalelement. Dann ist α auch eine Näherung für den größten Eigenwert λ und der erste Einheitsvektor e_1 eine Näherung für den zugehörigen Eigenvektor u. Damit ergibt sich als Eigenwertproblem:

$$A \begin{pmatrix} 1 \\ z \end{pmatrix} = \begin{pmatrix} \alpha & c^T \\ b & F \end{pmatrix} \begin{pmatrix} 1 \\ z \end{pmatrix} = \lambda \begin{pmatrix} 1 \\ z \end{pmatrix} \tag{4.2}$$

Dabei sind F eine quadratische Matrix, α ein Skalar und b, c und z Vektoren der entsprechenden Größe. Gesucht sind nun der Eigenwert λ, der in gewisser Weise nahe bei α liegt, und der zugehörige Eigenvektor

$$u = \begin{pmatrix} 1 \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ z \end{pmatrix} + e_1$$

Das Eigenwertproblem lässt sich anders schreiben als:

$$\begin{aligned} \lambda &= \alpha + c^T z \\ (F - \lambda I)z &= -b \end{aligned}$$

Die zweite Gleichung lässt sich mit D als Diagonale von F anders darstellen als:

$$\begin{aligned} & (F - \lambda I)z = -b \\ \Leftrightarrow \quad & (D - D + F - \lambda I)z = -b \\ \Leftrightarrow \quad & (D - \lambda I)z = (D - F)z - b \end{aligned}$$

Daraus lässt sich, mit θ_k als Näherung für λ und z_k als Näherung für z, die folgende Gauß-Jacobi-Iterationsvorschrift herleiten:

$$\begin{cases} \theta_k &= \alpha + c^T z_k \\ (D - \theta_k I)z_{k+1} &= (D - F)z_k - b \end{cases}$$

4.3 Das Jacobi-Davidson-Verfahren

Das Jacobi und das Davidson-Verfahren versuchen, eine gegebene Eigenvektornäherung zu korrigieren. Man sucht nun das orthogonale Komplement zur aktuellen Näherung u_k des gesuchten Eigenvektors u von A.

Jacobi schlug vor, die Komponente des gesuchten Eigenvektors senkrecht auf den Startvektor e_1 zu berechnen und betrachtete zu diesem Zweck nur noch das System (4.2) ohne die erste Zeile und Spalte. Dies entspricht einer Projektion des gegebenen Eigenwertproblems auf das orthogonale Komplement von e_1. Da man an der Komponente von x senkrecht auf die momentane Näherung u_k für x interessiert ist, beschränkt man sich auf den Unterraum $u_k^\perp$, der nur die Komponenten, die senkrecht auf u_k stehen, enthält..

Die orthogonale Projektion von A auf den Unterraum $u_k^\perp$ ist gegeben durch:

$$B = (I - u_k u_k^*)A(I - u_k u_k^*) \tag{4.3}$$

Dabei nimmt man an, dass u_k normalisiert ist. Für den Ritzwert θ_k gilt:

$$\begin{aligned} & \theta_k u_k = A u_k \\ \Leftrightarrow \quad & \theta_k u_k^* u_k = u_k^* A u_k \\ \Leftrightarrow \quad & \theta_k = u_k^* A u_k \end{aligned}$$

Daher lässt sich die Projektionsvorschrift (4.3) wie folgt umformen:

$$\begin{aligned} & B = (I - u_k u_k^*)A(I - u_k u_k^*) \\ \Leftrightarrow \quad & B = (A - u_k u_k^* A)(I - u_k u_k^*) \\ \Leftrightarrow \quad & B = A - u_k u_k^* A - A u_k u_k^* + u_k (u_k^* A u_k) u_k^* \\ \Leftrightarrow \quad & A = B + u_k u_k^* A + A u_k u_k^* - \theta_k u_k u_k^* \end{aligned} \tag{4.4}$$

Möchte man nun das Eigenpaar (λ, x) von A berechnen, so weiß man, dass θ_k nahe bei λ und u_k nahe bei x liegen. Weiterhin erfüllen ν mit $\nu \perp u_k$ und $x = u_k + \nu$ offensichtlich die Bedingung:

$$A(u_k + \nu) = \lambda(u_k + \nu) \tag{4.5}$$

Weiterhin gilt für die Matrix B:

$$Bu_k = (I - u_k u_k^*)A(I - u_k u_k^*)u_k = (I - u_k u_k^*)A(u_k - u_k) = 0$$

Damit folgt dann aus (4.5) durch Einsetzen von (4.4):

$$\begin{aligned}
& (B + u_k u_k^* A + A u_k u_k^* - \theta_k u_k u_k^*)(u_k + \nu) = \lambda(u_k + \nu) \\
\Leftrightarrow \quad & 0 + B\nu + u_k\theta_k + u_k < u_k, A\nu > + A u_k + 0 - \theta_k u_k + 0 = \lambda u_k + \lambda\nu \\
\Leftrightarrow \quad & B\nu - \lambda\nu = -(Au_k - \theta_k u_k) + (\lambda - \theta_k - < u_k, A\nu >)u_k \\
\Leftrightarrow \quad & (B - \lambda I)\nu = -r_k + (\lambda - \theta_k - < u_k, A\nu >)u_k \qquad (4.6)
\end{aligned}$$

Da die linke Seite und r_k beide senkrecht zu u_k sind, muss der Faktor für u_k verschwinden. Es gilt also:

$$(B - \lambda I)\nu = -r_k = (A - \theta_k I)u_k$$

Da λ unbekannt ist, ersetzt man es durch die momentane Näherung θ_k. Falls θ_k nicht nahe genug am gesuchten Eigenwert liegt, ist es aber häufig effizienter, λ durch einen festen Wert zu ersetzen, der nahe beim gesuchten Eigenwert liegt. Dadurch wird eine Konvergenz gegen einen nicht gewünschten Eigenwert oder eine lokale Stagnation verhindert. Dies führt mit Hilfe von (4.3) zu der Korrekturgleichung:

$$\begin{aligned}
& [(I - u_k u_k^*)A(I - u_k u_k^*) - \theta_k I]\nu = -r_k \\
\Leftrightarrow \quad & [(I - u_k u_k^*)A(I - u_k u_k^*)\nu - \theta_k\nu] = -r_k \\
\Leftrightarrow \quad & [(I - u_k u_k^*)A(I - u_k u_k^*)\nu - \theta_k(I - u_k u_k^*)\nu] = -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^*)[A(I - u_k u_k^*)\nu - \theta_k\nu] = -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^*)(A\nu - \theta_k\nu) = -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^*)(A - \theta_k I)\nu = -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^*)(A - \theta_k I)(I - u_k u_k^*)\nu = -r_k \qquad (4.7)
\end{aligned}$$

Diese Korrekturgleichung bildet die Grundlage für das von *Sleijpen* und *Van der Vorst* entwickelte Jacobi-Davidson-Verfahren. Für einen gegebenen Ritzwert θ_k wird aus (4.7) ein ν berechnet, mit dem der Unterraum erweitert wird. Dann berechnet man ein neues Ritzpaar in Bezug auf den erweiterten Unterraum. Diese Schritte werden wiederholt, bis die Eigenwert-Näherung genau genug ist. Da es viel zu aufwändig ist die Korrekturgleichung exakt zu lösen und diese durch die Ersetzung von λ durch θ_k in sich schon nicht exakt ist, ersetzt man die Matrix $A - \theta_k I$ durch eine Näherung K.

Gleichung (4.7) lässt sich bei genauerer Betrachtung noch weiter vereinfachen:

$$\begin{aligned}
& (I - u_k u_k^*)K(I - u_k u_k^*)\nu = -r_k \\
\Leftrightarrow \quad & (I - u_k u_k^*)K\nu = -r_k \\
\Leftrightarrow \quad & K\nu = -r_k + \alpha u_k \text{ mit } \alpha = u_k^*(A - \theta_k I)\nu
\end{aligned}$$

Da ν unbekannt ist, kann man zunächst auch α nicht berechnen. Durch einige Umformungen erhält man aber, unter Beachtung, dass $u_k \perp \nu$ ist, eine Berechnungsvorschrift für α:

$$\begin{aligned}
& \nu = -K^{-1}r_k + \alpha K^{-1}u_k \\
\Leftrightarrow \quad & 0 = -u_k^* K^{-1} r_k + \alpha u_k^* K^{-1} u_k \\
\Leftrightarrow \quad & \alpha = \frac{u_k^* K^{-1} r_k}{u_k^* K^{-1} u_k}
\end{aligned}$$

Dies sieht nun zunächst sehr aufwändig aus, da zwei Operationen mit K^{-1} durchgeführt werden müssen. Es wird sich aber später zeigen, dass dies, wenn man (4.7) mit einem iterativen Löser vorkonditioniert, auf eine Berechnung mit K^{-1} reduziert werden kann. Bei einem direkten Löser, bei dem eine Zerlegung durchgeführt wird, macht es auch keinen großen Unterschied, ob die bestimmte Zerlegung auf eine oder zwei rechte Seiten angewendet wird.

Geht man nun mal von $K = A - \theta_k I$ aus, so erhält mal formal:

$$\begin{aligned} & (A - \theta_k I)\nu = -r_k + \alpha u_k \\ \Leftrightarrow \quad & \nu = -(A - \theta_k I)^{-1} r_k + \alpha (A - \theta_k I)^{-1} u_k \\ & \quad = -u_k + \alpha (A - \theta_k I)^{-1} u_k \end{aligned}$$

Da u_k bereits im Unterraum liegt, wird dieser effektiv durch den normalisierten Vektor $(A - \theta_k I)^{-1} u_k$ erweitert.

Dies ist derselbe Vektor, der von der Rayleigh-Quotienten Iteration erzeugt wird. Bei exakter Arithmetik erhält man also eine beschleunigte Rayleigh Quotienten Iteration (RQI) für die invertierte charakteristische Matrix. Von dieser Methode ist bekannt, dass sie für symmetrische Matrizen kubische und für unsymmetrische quadratische Konvergenz aufweist. Die quadratische Konvergenz des Jacobi-Davidson-Verfahrens lässt sich auch erkennen, wenn man es als abgeleitete Newton-Iteration betrachtet (siehe: Sleijpen und Van der Vorst [17]). Da das RQI-Verfahren sehr schnell konvergiert, kann man keine sehr große Beschleunigung erwarten. Das JD-Verfahren kann aber noch effektiver beschleunigt werden, wenn die Korrekturgleichung (4.7) nur näherungsweise gelöst wird. Es geht dabei allerdings in der Regel die kubische beziehungsweise quadratische Konvergenz verloren.

Zusammenfassend kann man das Jacobi-Davidson-Verfahren als Kombination zweier Grundideen beschreiben:

- Zunächst wird mit Hilfe der Ritz-Galerkin Bedingung (siehe Gleichung (4.1) auf Seite 18) eine Näherung für das Eigenwertproblem $Ax = \lambda x$ im aktuellen Unterraum bestimmt. Als Lösung erhält man k Paare $(\theta_j^{(k)}, u_j^{(k)} := U_k s_j^{(k)})$, die Ritzwerte und Ritzvektoren von A bezüglich des Unterraums.
- Mit Hilfe der Korrekturgleichung (4.7) wird dann das orthogonale Komplement zum aktuellen Ritzvektor bestimmt, mit dem dann der Unterraum erweitert wird.

Nun lässt sich auch die Begriffsbildung erklären: Im Verfahren wird Jacobis Ansatz für die Suche nach dem orthogonalen Komplement verwendet. Weiterhin bedient man sich des Davidson-Verfahrens, um die Erweiterung des Unterraums durchzuführen.

Zu beachten bleibt aber, dass man die Korrekturgleichung (4.7) nicht notwendigerweise mit dem von Davidson verwendeten „Shift-and-Invert“Verfahren lösen muss. Dies bedeutet weiterhin, dass man nicht mehr länger auf diagonal dominante Matrizen beschränkt ist.

Löst man die Korrekturgleichung (4.7) näherungsweise, so läuft dies formal für spezielle Näherungen auf bereits bekannte Verfahren hinaus:

- Nimmt man die sehr grobe Näherung $\nu = -r_k$, so ist das Verfahren äquivalent zur Arnoldi-Methode beziehungsweise für symmetrisches A zum Lanczos-Verfahren.
- Nähert man den Operator $A - \theta_k I$ durch $D_A - \theta_k I$ an und vernachlässigt die Orthogonalitätsbedingung $\nu \perp u_k$, so erhält man das Davidson-Verfahren.

In der Abbildung 4.2 wird das Jacobi-Davidson-Verfahren im Detail beschrieben. Es handelt sich hier um eine noch sehr einfache Form des Algorithmus, bei der nur der größte Eigenwert λ_{max} von A gesucht wird. Auf verfeinerte Algorithmen wird an späterer Stelle genauer eingegangen.

setze t auf Startnäherung $t = v_0$
for $m = 1, ...$
  for $i = 1, ..., m-1$
    $t = t - < v_i, t > v_i$
  end for
  $v_m = \frac{t}{\|t\|_2}$
  $v_m^A = Av_m$
  for $i = 1, ..., m-1$
    $M_{i,m} = v_i^* v_m^A$
    $M_{m,i} = v_m^* v_i^A$
  end for
  $M_{m,m} = v_m^* v_m^A$
  berechne das größte Eigenpaar $Ms = \theta s$ von M mit $\|s\|_2 = 1$
  $u = Vs$ mit $V = (v_1, ..., v_n)$
  $u^A = V^A s$ mit $V^A = (v_1^A, ..., v_n^A)$
  $r = u^A - \theta u$
  if $\|r\|_2 \leq \epsilon$ **then**
    $\tilde{\lambda} = \theta$
    $\tilde{x} = u$
    stop
  end if
  berechne näherungsweise $t \perp u$ aus $(I - uu^*)(A - \theta I)(I - uu^*)t = -r$
end for

Abbildung 4.2: einfacher Jacobi-Davidson-Algorithmus für λ_{max} von A

4.4 Iterative Lösung der Korrekturgleichung

Sei K ein Vorkonditionierer für die Matrix $A - \theta_j^{(k)} I$, so dass gilt:

$$K^{-1}(A - \theta_j^{(k)} I) \approx I$$

Natürlich muss der Operator K auch auf den Unterraum senkrecht zu $u_j^{(k)}$ beschränkt sein, weshalb man eigentlich mit folgendem Vorkonditionierer arbeitet:

$$\tilde{K} := (I - u_j^{(k)} u_j^{(k)^*}) K (I - u_j^{(k)} u_j^{(k)^*})$$

Auf den ersten Blick hin sieht das nach viel Overhead aus, da die aufwändigen Projektionen $I - u_j^{(k)} u_j^{(k)^*}$ durchgeführt werden. Aber dies kann auf einige einfache Operationen zurückgeführt werden, wie nun gezeigt wird.

Es wird im Folgenden davon ausgegangen, dass ein Krylov-Löser mit linksseitigem Vorkonditionierer und Startwert $t_0 = 0$ für die Lösung der Korrekturgleichung (4.7) verwendet wird. Da der Startvektor im Unterraum liegt, werden auch alle Iterationsvektoren des Krylov-Lösers im Unterraum liegen. Im jeweiligen Unterraum berechnet man nun den Vektor $z := \tilde{K}^{-1} \tilde{A} \nu$ für einen Vektor ν aus dem Krylov-Unterraum und die Matrix

$$\tilde{A} = (I - u_j^{(k)} u_j^{(k)^*})(A - \theta_j^{(k)} I)(I - u_j^{(k)} u_j^{(k)^*})$$

Dies geschieht nun in zwei Schritten: Zunächst berechnet man:

$$\begin{aligned} \tilde{A}\nu &= (I - u_j^{(k)} u_j^{(k)^*})(A - \theta_j^{(k)} I)(I - u_j^{(k)} u_j^{(k)^*})\nu \\ &= (I - u_j^{(k)} u_j^{(k)^*}) y \text{ mit } y = (A - \theta_j^{(k)} I)\nu \end{aligned}$$

Nun lässt sich also $z \perp u_j^{(k)}$ berechnen durch:

$$\tilde{K}z = (I - u_j^{(k)} u_j^{(k)^*})y$$

Dies lässt sich anders schreiben als:

$$\begin{aligned} & (I - u_j^{(k)} u_j^{(k)^*})K(I - u_j^{(k)} u_j^{(k)^*})z = (I - u_j^{(k)} u_j^{(k)^*})y \\ \Leftrightarrow \quad & K(I - u_j^{(k)} u_j^{(k)^*})z = y \\ \Leftrightarrow \quad & Kz = y - \alpha u_j^{(k)} \end{aligned}$$

Ein paar Umformungen führen weiter zu einer Berechungsvorschrift für α:

$$\begin{aligned} & z = K^{-1}y - \alpha K^{-1} u_j^{(k)} \\ \Leftrightarrow \quad & 0 = u_j^{(k)^*} K^{-1} y - \alpha u_j^{(k)^*} K^{-1} u_j^{(k)} \\ \Leftrightarrow \quad & \alpha = \frac{u_j^{(k)^*} K^{-1} y}{u_j^{(k)^*} K^{-1} u_j^{(k)}} \end{aligned}$$

Bei i_s Iterationen des linearen Lösers müssen nun nur $i_s + 1$ Operationen mit K^{-1} durchgeführt werden, da nur einmal pro Iteration $K^{-1}u_j^{(k)}$ berechnet werden muss. Weiterhin benötigt man für eine Matrix-Vektor-Multiplikation $\tilde{K}^{-1}\tilde{A}$ nur ein inneres Produkt und ein Vektor-Update anstatt vier Berechnungen mit dem Projektor $I - u_j^{(k)} u_j^{(k)^*}$.
In der folgenden Abbildung ist das Verfahren zur Lösung der Korrekturgleichung (4.7) mit einer Krylov-Unterraum-Methode und Vorkonditionierung im Detail beschrieben.

$\hat{u} = K^{-1}u$
$\mu = u^*\hat{u}$
$\hat{r} = K^{-1}r$
$\tilde{r} = \hat{r} - \frac{u^*\hat{r}}{\mu}$
berechne durch Krylov-Löser $\tilde{K}^{-1}\tilde{A}\nu = -\tilde{r}$
$y = (A - \theta I)\nu$
$\hat{y} = K^{-1}y$
$z = \hat{y} - \frac{u^*\hat{y}}{\mu}\hat{u}$

Abbildung 4.3: Lösung der Korrekturgleichung

4.5 Restart-Strategien

Für ein Eigenpaar

Bei der Vergrößerung des Unterraums entsteht ein großer Overhead an Speicherplatz und Rechenzeit, da mit der Zeit wesentlich größere Systeme bearbeitet werden müssen. Dieser Overhead macht einen so genannten Restart notwendig. Ein offensichtlicher Weg für einen Restart ist, die letzten Näherungen für den gewünschten Eigenvektor zu nehmen, auch wenn dies nicht notwendigerweise die effizienteste Methode ist. Mit jedem Restart lässt man wertvolle Informationen über den Unterraum fallen, die im verbleibenden Teil des Unterraumes enthalten ist. Trotzdem hat man einen brauchbaren Unterraum, in dem alle Vektoren Informationen über den gewünschten Eigenvektor enthalten.

Der Informationsverlust bei einem Restart äußert sich im Verlust an Konvergenzgeschwindigkeit nach dem Restart. Daher ist es oft besser, mit einer Menge von Eigenvektoren den Restart durchzuführen, die einen Unterraum aufspannen, der mehr Informationen für das gewünschte Eigenpaar

enthält. Eine gute Strategie ist hier, den Restart mit dem Unterraum durchzuführen, der durch eine kleine Menge von Ritzvektoren aufgespannt wird, die am nächsten an dem gegebenen Zielvektor liegen.

Für mehrere Eigenpaare

Unter Umständen hat das Jacobi-Davidson-Verfahren einige Nachteile gegenüber dem Standard Arnoldi-Verfahren. Das Jacobi-Davidson-Verfahren konvergiert sehr schnell gegen einen Eigenwert. Für das Arnoldi-Verfahren ist es kein großes Problem, auch mehrere Eigenwerte zu berechnen, da die gewöhnlich langsamere Konvergenz von Ritzwerten zu einem speziellen Eigenwert Hand in Hand geht mit der gleichzeitigen Konvergenz von anderen Ritzwerten zu anderen Eigenwerten. Jacobi-Davidson zielt absichtlich nur auf einen speziellen Eigenwert ab, der am nächsten bei einem vorgegebenen Ziel liegt. Das Arnoldi-Verfahren hingegen führt zu Näherungen für eventuell alle Eigenwerte. In der frühen Phase der Iteration führt es gewöhnlich zu isolierten äußeren Eigenwerten.

Für das Jacobi-Davidson-Verfahren würde man einen Restart mit unterschiedlichen ausgewählten Ritzpaaren durchführen. Es ist nicht garantiert, dass dies zu einem neuen Eigenpaar führt. Ein weiteres Problem ist die Entdeckung von mehrfachen Eigenwerten, aber dieses Problem besteht auch bei anderen Unterraumverfahren.

Ein bekannter Weg, um dieses Problem zu beheben, ist die *Deflation*-Strategie. Konvergiert das Verfahren gegen einen Eigenvektor, so wird die Iteration in einem Unterraum fortgeführt, der von den restlichen Eigenvektoren aufgespannt wird. Für hermitesche Matrizen A stellt dies kein Problem dar, da sie ein orthonormales Eigenvektorsystem besitzen. Da es verhindert werden soll, dass mit nicht orthogonalen Reduktions- beziehungsweise Deflationsoperatoren gearbeitet wird, sollte man für nicht-hermitesche Matrizen mit so genannten *Schurvektoren* arbeiten. Diese Schurvektoren sind definiert als die Spalten einer orthogonalen Matrix Q, die A in eine obere Dreiecksform transformiert:

$$Q^*AQ = R$$

Dadurch erhält man die Eigenwerte von A auf der Diagonalen von R und die Eigenvektoren x von A können aus den Eigenvektoren y von R durch $x = Qy$ berechnet werden.

Es existiert ein Algorithmus zur Berechnung mehrerer Eigenpaare. Dieser basiert auf der Berechnung einer Schurform von A:

$$AQ_k = Q_kR_k$$

Dabei ist Q_k eine orthonormale $n \times k$ Matrix, R_k eine obere $k \times k$ Dreiecksmatrix, wobei $k << n$ gilt. Ist nun (x, λ) ein Eigenpaar von R_k, so ist (Q_kx, λ) ein Eigenpaar von A.

Zur Berechnung der Schurform für Eigenwerte nahe einem bestimmten Wert τ benötigt man folgende Schritte:

1. Gegeben sei ein orthonormales Unterraumbasissystem $v_1, ..., v_i$ mit zugehöriger Matrix V_i. Dann berechnet sich die Projektionsmatrix $M = V_i^*AV_i$. Dabei ist M eine $i \times i$ Matrix. Für M berechnet man dann die vollständige Schurform $MU = US$, wobei $U^*U = I$ gilt und S eine obere Dreiecksmatrix ist. Dies kann man zum Beispiel über eine Standard QR-Zerlegung machen.

   Dann ordnet man S so um, dass $|S_{i,i} - \tau|$ eine aufsteigende Reihe für wachsendes i bildet. Die ersten Diagonalelemente von S repräsentieren dabei Eigenwertnäherungen nahe an τ und die ersten entsprechenden Spalten von V_i stellen den Unterraum der besten Eigenvektornäherungen dar. Die Umordnung kann unter Beibehaltung der Dreiecksgestalt von S erfolgen.

   Steht nur begrenzter Speicherplatz zur Verfügung, so kann man diesen Unterraum auch für den Restart nutzen, während man die restlichen Spalten einfach ignoriert. Der restliche Un-

terraum wird wie beim Jacobi-Davidson-Verfahren erweitert. Nachdem das Verfahren konvergiert ist, erhält man ein Eigenpaar (q, λ) von A: $Aq = \lambda q$.

2. Angenommen es ist bereits eine Schurform der Dimension k vorhanden und diese soll mit einer neuen Spalte q erweitert werden:

$$A[Q_k, q] = [Q_k, q] \begin{bmatrix} R_k & s \\ & \lambda \end{bmatrix}$$

Dabei ist $Q_k^* q = 0$. Nach einigen Berechnungen erhält man hier:

$$(I - Q_k Q_k^*)(A - \lambda I)(I - Q_k Q_k^*)q = 0$$

Dies sagt aus, dass ein neues Paar (q, λ) ein Eigenpaar von

$$\tilde{A} = (I - Q_k Q_k^*)A(I - Q_k Q_k^*)$$

ist. Dieses Paar kann man durch Anwendung des Jacobi-Davidson-Verfahrens (mit Schurform Reduktion, wie in Schritt 1 für $\tilde{A}$) lösen.

Man sieht, dass nach jeder Konvergenz gegen ein Eigenpaar die Matrix $\tilde{A}$ zu aufwändigeren Berechnungen führt, aber es lässt sich zeigen, dass das gesamte Verfahren zu einem sehr effizienten Berechnungsprozess führt. Eine Erklärung hierfür ist, dass nach der Konvergenz einiger Eigenvektoren, die zu Eigenwerten nahe dem Ziel τ gehören, die Matrix $\tilde{A}$ besser konditioniert ist. Das bedeutet, dass die Korrekturgleichung im Jacobi-Davidson-Verfahren einfacher von einem iterativen Löser gelöst werden kann.

Diese Korrekturgleichung kann mit einem vorkonditionierten iterativen Löser gelöst werden. Derselbe Vorkonditionierer kann auch für die verschiedenen Eigenpaare verwendet werden. Daher lohnt es sich, gute Vorkonditionierer zu entwickeln.

Mit der zuvor erwähnten Restart-Strategie kann man erwarten, dass, wenn ein Ritzwert nahe genug an einem Eigenwert liegt, der verbleibende Teil des Unterraums schon viele Komponenten in der Nähe der Eigenpaare besitzt, da man in jedem Schritt die Ritzvektoren, die nahe an dem gewünschten Eigenwert lagen, genommen hat. Diese Information kann man als Basis für einen Unterraum zur Berechnung des nächsten Eigenvektors nutzen, nachdem die Eigenwertgleichung geeignet umgeformt wurde, um zu verhindern, dass der gerade gefundene Eigenvektor in den Berechnungsprozess wieder mit eingeht.

Seien nun $q_1, ..., q_{k-1}$ die akzeptierten Schurvektoren und seien nun diese Vektoren in guter Genauigkeit orthonormal. Dieses Verfahren mit genäherten Schurvektoren mit Residuum $\|Aq_j - QRe_j\|_2$ innerhalb einer Toleranzgrenze ϵ, kann einen Fehler der Ordnung von ϵ^2 in den Eigenwerten hervorrufen. Voraussetzung dabei ist, dass die berechneten Eigenwerte getrennt von den restlichen Eigenwerten liegen.

Die Matrix Q_{k-1} hat die Spaltenvektoren q_j. Daher wendet man den Jacobi-Davidson-Algorithmus auf die Matrix

$$(I - Q_{k-1} Q_{k-1}^*)A(I - Q_{k-1} Q_{k-1}^*)$$

an, um den nächsten Schurvektor q_k zu erhalten. Dies führt zu einer Korrekturgleichung der Form:

$$P_m(I - Q_{k-1} Q_{k-1}^*)(A - \theta_j^{(m)} I)(I - Q_{k-1} Q_{k-1}^*)P_m t_j^{(m)} = -r_j^{(m)}$$

Dabei ist $P_m := I - u_j^{(m)} u_j^{(m)*}$. Dies muss für die Korrektur $t_j^{(m)}$ für jede neue Schurnäherung $u_j^{(m)}$ mit zugehörigem Ritzwert $\theta_j^{(m)}$ gelöst werden.

4.6 Vorkonditionieren

Sei nun K ein Vorkonditionierer von links für den Operator $A - \theta_j^{(m)} I$ und $\tilde{Q}$ die Matrix Q_{k-1} erweitert um die k-te Spalte $u_k^{(m)}$. In diesem Fall muss der Vorkonditionierer K auf den Unterraum orthogonal zu $\tilde{Q}$ beschränkt sein, was bedeutet, dass man effektiv mit

$$\tilde{K} = (I - \tilde{Q}\tilde{Q}^*)K(I - \tilde{Q}\tilde{Q}^*)$$

arbeitet. Im Folgenden wird nun davon ausgegangen, dass ein Krylov-Löser mit Startwert $t_0 = 0$ und ein Linksvorkonditionierer für die Näherungslösung der Korrekturgleichung verwendet wird. Da der Startvektor im Unterraum orthogonal zu $\tilde{Q}$ ist, sind alle Iterationsvektoren des Krylov-Lösers auch in diesem Unterraum. In diesem Unterraum muss der Vektor $z := \tilde{K}^{-1}\tilde{A}v$ berechnet werden. Dabei wird der Vektor v durch den Krylov-Löser bereitgestellt. Weiterhin gilt:

$$\tilde{A} := (I - \tilde{Q}\tilde{Q}^*)(A - \theta_j^{(m)} I)(I - \tilde{Q}\tilde{Q}^*)$$

Dies geschieht in zwei Schritten. Erst berechnet man:

$$\begin{aligned}\tilde{A}v &= (I - \tilde{Q}\tilde{Q}^*)(A - \theta_j^{(m)} I)(I - \tilde{Q}\tilde{Q}^*)v \\ &= (I - \tilde{Q}\tilde{Q}^*)y \text{ mit } y := (A - \theta_j^{(m)} I)v\end{aligned}$$

Danach löst man mit Links-Vorkonditionieren $z \perp \tilde{Q}$ aus

$$\tilde{K}z = (I - \tilde{Q}\tilde{Q}^*)y$$

Da $\tilde{Q}z = 0$, folgt daraus

$$Kz = y - \tilde{Q}\alpha \quad \Leftrightarrow \quad z = K^{-1}y - K^{-1}\tilde{Q}\alpha$$

Die Bedingung $\tilde{Q}^*z = 0$ führt zu

$$\alpha = (\tilde{Q}^*K^{-1}\tilde{Q})^{-1}\tilde{Q}^*K^{-1}y$$

Der Vektor $\hat{y} := K^{-1}y$ wird dabei gelöst durch $K\hat{y} = y$ und ebenso wird $\hat{Q} := K^{-1}\tilde{Q}$ durch $K\hat{Q} = \tilde{Q}$ gelöst.

Diese beiden Gleichungen müssen dabei nur einmal in einem Iterationsprozess gelöst werden. Daher sind effektiv nur $i_s + k$ Operationen mit dem Vorkonditionierer für i_s Iterationen mit dem linearen Löser erforderlich. Die Matrix-Vektor-Multiplikation mit dem links vorkonditionierenden Operator in einer Iteration mit dem Krylov-Löser erfordert nur eine Operation mit $\tilde{Q}^*$ und $K^{-1}\tilde{Q}$, anstatt der vier Mal für den Projektionsoperator $(I - \tilde{Q}\tilde{Q}^*)$.

Offensichtliche Einsparungen werden erzielt, wenn man den Operator K für eine Reihe von aufeinander folgenden Eigenwertberechnungen beibehält. In diesem Fall kann man die Spalten von $\hat{Q}$ aus den vorherigen Schritten übernehmen.

4.7 Der vollständige Algorithmus

Der Jacobi-Davidson-QR-Algorithmus beinhaltet die Möglichkeit eines Restarts, wenn die Dimension des gegenwärtigen Unterraums k_{max} überschreitet. Er berechnet k_{max} Eigenwerte nahe einem vorgegebenen Ziel τ in der komplexen Ebene. Hierbei muss man notwendigerweise ungenau sein, da die Eigenwerte einer gewöhnlichen nicht-hermiteschen Matrix in der komplexen Ebene nicht angeordnet sind. Welche Ritzwerte der Algorithmus als nahe Eigenwerte liefert, hängt unter anderem von den Winkeln der zugehörigen Ritzvektoren mit den gesuchten Eigenvektoren ab.

Gewöhnlich ist die Auswahl von Eigenwerten passend, wenn τ irgendwo aus dem äußeren Spektrum der Eigenwerte gewählt ist. Will man dagegen Eigenwerte von A aus dem Innern des Spektrums berechnen, so ist dieser Algorithmus wenig zufrieden stellend. In diesem Fall sollte man besser mit harmonischen Ritzwerten (siehe Kapitel 4.8) arbeiten.

Zur Anwendung des Algorithmus müssen eine Toleranzgrenze ϵ, das gewünschte Ziel τ und die Anzahl der Eigenpaare k_{max} die um τ herum berechnet werden sollen, vorgegeben werden. Dabei gibt k_{max} hierbei auch gleichzeitig die maximale Größe des Suchraums vor. Wird k_{max} überschritten, so findet ein Restart mit vorgegebener Dimension m_{min} statt. Nach Beendigung des Algorithmus erhält man k_{max} Eigenwerte nahe bei τ und die zugehörige Schurform $AQ = QR$, wobei Q eine $n \times k_{max}$ Orthogonalmatrix und R eine $k_{max} \times k_{max}$ obere Dreiecksmatrix ist. Die Eigenwerte findet man dabei auf der Diagonalen von R. Die berechnete Schurform genügt der Gleichung $\|Aq_j - QRe_j\|_2 \leq j\epsilon$, wobei q_j die j-te Spalte von Q bezeichnet.

Es folgen nun einige Bemerkungen zu den einzelnen Schritten des Algorithmus:

1. Der neue Vektor t wird in Bezug auf den vorliegenden Suchraum orthonormalisiert. Die Orthonormalisierung kann zum Beispiel mit Hilfe des Gram-Schmidt-Verfahren durchgeführt werden. Eine andere Möglichkeit ist das iterative modifizierte Gram-Schmidt-Verfahren (siehe Abbildung 3.1). Dies verbessert die numerische Stabilität.
2. Hier wird die letzte Spalte und die letzte Zeile der dicht besetzten Matrix $M := V^*AV = V^*V^A$ der Ordnung m berechnet. Dabei ist $V^A := AV$ und V eine $n \times m$ Matrix mit den Spalten v_j. Das gleiche gilt für V^A.
3. Die Schurzerlegung der $m \times m$ Matrix M kann durch einen Standardlöser für dichte Eigenwertprobleme gelöst werden. In diesem Algorithmus werden die Standard Ritzwerte berechnet, weshalb sich der Algorithmus zur Berechnung von k_{max} äußeren Eigenwerten von A in der Nähe von τ eignet. Zur Berechnung von inneren Eigenwerten sollte man einen Algorithmus mit harmonischen Ritzwerten verwenden.

   In jedem Schritt muss die Schurform so sortiert werden, dass $T_{1,1}$ am nächsten an τ liegt. Eine Ausnahme stellt hierbei der Fall dar, dass $m \geq m_{max}$ ist, denn dann sind alle m_{max} führenden Diagonalelemente am nächsten bei τ, so dass in diesem Fall die Sortierung übersprungen werden kann.
4. Eine Schurvektornäherung wird dann akzeptiert, wenn die Norm des Residuums kleiner als ϵ ist. Das bedeutet, dass Ungenauigkeiten in der Größenordnung von ϵ in den berechneten Eigenwerten akzeptiert werden. Die Ungenauigkeiten der Schurvektoren (in den Winkeln) liegen ebenfalls in der Größenordnung $\mathcal{O}(\epsilon)$, falls der betreffende Eigenwert einfach ist und separiert von den anderen Eigenwerten liegt.

   Handelt es sich bei der Matrix um eine reelle Matrix, so sind die Eigenwerte entweder auch alle reell oder sie kommen immer konjugiert komplex vor. Wird also ein komplexer Eigenwert gefunden, so kennt man den konjugiert komplexen und kann dies ausnutzen, was den Algorithmus effizienter macht.
5. Wird ein Ritzpaar akzeptiert, so wird mit der Suche nach dem nächsten Paar fortgefahren. Dabei bilden die verbleibenden Ritzvektoren die Basis für den neuen Suchraum.
6. Überschreitet die gerade aktuelle Dimension des Suchraums m_{max}, so wird ein Restart durchgeführt. Dabei werden die m_{min} Ritzvektoren, deren zugehöriger Ritzwert am nächsten bei τ liegt, als Spannvektoren des neuen Suchraums verwendet.
7. In der Matrix Q befinden sich die berechneten Schurvektoren. Die Matrix $\tilde{Q}$ besteht aus der Matrix Q, erweitert um den gerade aktuellen Schurvektor u. So erhält man eine kompaktere Schreibweise. Die zu lösende Korrekturgleichung ist äquivalent zu
   $P_m(I - Q_{k-1}Q_{k-1}^*)(A - \theta_j^{(m)}I)(I - Q_{k-1}Q_{k-1}^*)P_m t_j^{(m)} = -r_j^{(m)}$.
   Die neue Korrektur t muss dabei sowohl orthogonal zu den Spalten von Q als auch zu u sein.

   Die Korrekturgleichung kann nun mit jedem passenden Verfahren gelöst werden, zum Bei-

spiel mit vorkonditionierten Krylov-Verfahren, die für unsymmetrische Systeme gedacht sind. Es muss sichergestellt werden, dass der Startvektor t_0 für den iterativen Löser die Orthogonalitätsbedingung $\tilde{Q}^* t_0 = 0$ erfüllt.

Die Korrekturgleichung muss dabei nicht sehr genau gelöst werden. Es ist hier sinnvoll, wie es oft für inexakte Newton-Verfahren benutzt wird, die Genauigkeit mit jedem Iterationsschritt des Jacobi-Davidson-Verfahrens zu erhöhen. Man könnte zum Beispiel die Korrekturgleichung im j-ten Iterationsschritt mit einer Residuumminimierung von 2^{-j} lösen. Findet man einen Schurvektor, so wird j wieder auf 0 zurückgesetzt. Diese Strategie macht Sinn, da man das Jacobi-Davidson-Verfahren auch als ein (inexaktes) Newton-Verfahren bezeichnen kann.

In den ersten Schritten macht es wenig Sinn, die Korrekturgleichung exakt zu lösen, da der genäherte Eigenwert θ noch sehr ungenau sein wird. Hier kann es sogar effektiver sein, θ vorübergehend durch τ zu ersetzen oder $t = -r$ für die Erweiterung des Suchraums zu benutzen.

$t = v_0, k = 0, m = 0; Q = [\,], R = [\,];$
while $k < k_{max}$
(1) **for** $i = 1, .., m$
$t = t - (v_i^* t) v_i$
end for
$m = m + 1$, $v_m = \frac{t}{\|t\|_2}$, $v_m^A = A v_m$
(2) **for** $i = 1, .., m - 1$
$M(i, m) = v_i^* v_m^A$, $M(m, i) = v_m^* v_i^A$
end for
$M(m, m) = v_m^* v_m^A$
(3) Schurzerlegung $M = STS^*$, mit S orthogonal und
T obere Dreiecksmatrix, mit $|T_{i,i} - \tau| \leq |T_{i+1,i+1} - \tau|$
$x = Sy, u = Vx_1, u^A = V^A x_1, \theta = T_{1,1}, r = u^A - \theta u, \tilde{a} = Q^* r, \tilde{r} = r - Q\tilde{a}$
(4) **while** $\|\tilde{r}\|_2 \leq \epsilon$
$R = \begin{pmatrix} R & \tilde{a} \\ 0 & \theta \end{pmatrix}$
$Q = [Q, u]$, $k = k + 1$
if $k = k_{max}$ **then**
stop
end if
$m = m - 1$
(5) **for** $i = 1, .., m$
$v_i = V x_{i+1}$, $v_i^A = V_{x_{i+1}}^A$, $x_i = e_i$
end for
M = unterer $m \times m$ Block von T
$u = v_1$, $\theta = M_{1,1}$, $r = v_1^A - \theta u$, $\tilde{a} = Q^* r$, $\tilde{r} = r - Q\tilde{a}$
end while
(6) **if** $m \geq m_{max}$ **then**
for $i = 2, .., m_{min}$
$v_i = V x_i$, $v_i^A = V^A x_i$
end for
M = führender $m_{min} \times m_{min}$ Block von T
$v_1 = u$, $v_1^A = u^A$, $m = m_{min}$
end if
$\tilde{Q} = [Q, u]$
(7) Löse $t(\perp \tilde{Q})$ näherungsweise über : $(I - \tilde{Q}\tilde{Q}^*)(A - \theta I)(I - \tilde{Q}\tilde{Q}^*) = -\tilde{r}$
end while

Abbildung 4.4: Jacobi-Davidson-Verfahren zur Berechnung von k_{max} äußeren Eigenwerten

4.8 Harmonische Ritzwerte

Wie schon in den vorangegangenen Kapiteln erläutert wurde, generiert das Jacobi-Davidson-Verfahren Basisvektoren $v_1, ..., v_m$ für einen Unterraum $\mathcal{V}_m$. Für die Projektion der Matrix A in den Unterraum, müssen die Vektoren $w_j := Av_j$ berechnet werden. Die Inversen der Ritzwerte der Matrix A^{-1} in Bezug auf den Unterraum, der durch die Vektoren w_j aufgespannt wird, nennt man *harmonische Ritzwerte*. Man kann sie allerdings berechnen, ohne A explizit zu invertieren, da ein harmonisches Ritzpaar $(\tilde{\theta}_j^{(m)}, \tilde{u}_j^{(m)})$ die Bedingung

$$A\tilde{u}_j^{(m)} - \tilde{\theta}_j^{(m)}\tilde{u}_j^{(m)} \perp \mathcal{W}_m := \text{span}(w_1, ..., w_m)$$

erfüllt. Dabei ist $\tilde{u}_j^{(m)} \epsilon \mathcal{V}_m$ und $\tilde{u}_j^{(m)} \neq 0$. Dies beinhaltet, dass die harmonischen Ritzpaare Eigenwerte von $(W_m^* A V_m, W_m^* V_m)$ sind, das heißt es gilt:

$$W_m^* A V_m s_j^{(m)} - \tilde{\theta}_j^{(m)} W_m^* V_m s_j^{(m)} = 0$$

Die äußeren Standard-Ritzwerte konvergieren gewöhnlich gegen die äußeren Eigenwerte von A. Dementsprechend konvergieren die inneren harmonischen Ritzwerte der geshifteten Matrix $A - \tau I$ gewöhnlich gegen die Eigenwerte $\lambda \neq \tau$, die am nächsten bei dem Shift τ liegen. Glücklicherweise stimmen der Suchraum $\mathcal{V}_m$ der geshifteten und der ungeshifteten Matrix überein, wodurch die Berechnung von harmonischen Ritzpaaren vereinfacht wird. Aus Stabilitätsgründen konstruiert man $\mathcal{V}_m$ und $\mathcal{W}_m$ so, dass sie beide (näherungsweise) orthonormal sind; das heißt das $\mathcal{W}_m$ so konstruiert wird, dass $(A - \tau I)V_m = W_m M_m^A$, wobei M_m^A eine $m \times m$ obere Dreiecksmatrix ist.

Die harmonischen Ritzvektoren beinhalten meist mehr Informationen als die harmonischen Ritzwerte. Besonders dann, wenn der genäherte Eigenvektor einen kleinen Winkel mit dem Eigenvektor bildet und das Ziel τ sehr nahe am zugehörigen Eigenwert liegt, ist der Ritzwert nicht sehr gut. Der Grund dafür ist, dass ein harmonisches Eigenpaar $(\tilde{\theta}_j^{(m)}, \tilde{u}_j^{(m)})$ zu dem Residuum

$$r = A\tilde{u}_j^{(m)} - \tilde{\theta}_j^{(m)}\tilde{u}_j^{(m)}$$

führt und es gilt :

$$r \perp (A - \tau I)\tilde{u}_j^{(m)}$$

Wenn nun $\tilde{u}_j^{(m)}$ und τ beide eine gute Approximation für ein Eigenpaar von A sind, so kann $\tilde{\tau}_j^{(m)}$ nur eine schlechte Näherung werden. Deshalb hat man Restart-Strategien entwickelt, die die Rayleigh-Quotienten mit den harmonischen Ritzwerten verknüpfen. Eine detaillierte Herleitung zum Jacobi-Davidson-Algorithmus für harmonische Ritzwerte ist in [7] gegeben.

Es ist hier sinnvoller, eine Schurzerlegung durchzuführen statt einer Eigenwertzerlegung. Dabei werden zuerst Vektoren $q_1, ..., q_k$ berechnet, so dass $AQ_k = Q_k R_k$, wobei $Q_k^* Q_k = I_k$ gilt und R_k eine $k \times k$ obere Dreiecksmatrix ist. Die Diagonalelemente von R_k bilden die Eigenwerte von A. Die zugehörigen Eigenvektoren von A können dann über Q_k und R_k berechnet werden.

Das Verfahren der harmonischen Ritzwerte weist ein Problem auf: Der Raum W wird durch den Zusammenhang $W = (A - \tau I)V$ generiert. Je genauer τ an einem bestimmten Eigenwert liegt je besser kann dieser separiert werden. Allerdings wird die Kondition des Systems W mit zunehmender Genauigkeit von τ, bezüglich eines Eigenwertes, zunehmend schlechter. Weiterhin wird die Kondition zunehmend schlechter, wenn die Eigenvektornährungen besser werden. Insgesamt ist das harmonische Ritz-Verfahren wesentlich schlechter konditioniert als das Stardard-Ritz-Verfahren.

$t =$ random, $k = 0, m = 0, Q = [\,], R = [\,]$;
while $k < k_{max}$
$w = (A - \tau I)t$
(1) **for** $i = 1, .., m$
$\gamma = w_i^* w$, $w = w - \gamma * w_i$, $t = t - \gamma v_i$
end for
$m = m + 1$, $w_m = \frac{w}{\|w\|_2}$, $v_m = \frac{t}{\|w\|_2}$
(2) **for** $i = 1, .., m - 1$
$M_{i,m} = w_i^* v_m$, $M_{m,i} = w_m^* v_i$
end for
$M_{m,m} = w_m^* v_m$
(3) Schurzerlegung $M = STS^*$, mit S orthogonal und
T obere Dreiecksmatrix, mit $|T_{i,i} - \tau| \leq |T_{i+1,i+1} - \tau|$
$x = Sy$, $\tilde{u} = VS_1$, $\mu = \|\tilde{u}\|_2$,$u = \frac{\tilde{u}}{\mu^2}$, $\theta = \frac{T_{1,1}}{\mu^2}$, $\tilde{w} = WS_1$,$r = \frac{\tilde{w}}{\mu} - \theta u$,
$\tilde{a} = Q^* r$, $\tilde{r} = r - Q\tilde{a}$
(4) **while** $\|\tilde{r}\|_2 \leq \epsilon$
$R = \begin{pmatrix} R & \tilde{a} \\ 0 & \theta + \tau \end{pmatrix}$
$Q = [Q, u]$, $k = k + 1$
if $k = k_{max}$ **then**
stop
end if
$m = m - 1$
(5) **for** $i = 1, .., m$
$v_i = Vx_{i+1}$, $w_i = Wx_{i+1}$, $x_i = e_i$
end for
M = unterer $m \times m$ Block von T
$\mu = \|v_1\|_2$, $\theta = \frac{T_{2,2}}{\mu^2}$, $u = \frac{v_1}{\mu}$, $r = \frac{w_1}{\mu} - \theta u$, $\tilde{a} = Q^* r$, $\tilde{r} = r - Q\tilde{a}$
end while
(6) **if** $m \geq m_{max}$ **then**
for $i = 2, .., m_{min}$
$v_i = VS_i$, $w_i = WS_i$
end for
M = führender $m_{min} \times m_{min}$ Block von T
$v_1 = \tilde{u}$, $w_1 = \tilde{w}$, $m = m_{min}$
end if
$\tilde{Q} = [Q, u]$
(7) löse $t(\perp \tilde{Q})$ näherungsweise über : $(I - \tilde{Q}\tilde{Q}^*)(A - (\theta + \tau)I)(I - \tilde{Q}\tilde{Q}^*) = -\tilde{r}$
end while

Abbildung 4.5: Jacobi-Davidson-Verfahren zur Berechnung von k_{max} inneren Eigenwerten über harmonische Ritzwerte

Kapitel 5

Speicherung und Aufteilung der Matrizen

Im ersten Kapitel wurden verschiedene bekannte Besetzungsschemata von Matrizen mathematisch beschrieben. Die Hauptaufgabe der Diplomarbeit bestand darin, das bestehende Programm zu reorganisieren. Bei der Reorganisation war eines der Hauptziele, das Programm so zu strukturieren, dass es möglichst einfach wird, neue Matrix-Formate einzubinden. In der bestehenden Version der Programmes stand ausschließlich das CRS-Format (siehe Kapitel 5.1) zur Speicherung der Matrizen bereit. In der Diplomarbeit wurde zum Test der neu organisierten Struktur ein Format zum Speichern von Bandmatrizen (siehe Kapitel 5.5) eingefügt. Im Kapitel 7.8 wird beschrieben welche Schritte durchzuführen sind, um neue Matrix-Formate einzubinden. In den nächsten Kapiteln werden die beiden bestehenden Formate sehr detailliert erläutert.

5.1 Speicherung von dünnbesetzten Matrizen

Für dünnbesetzte Matrizen, so genannte Sparse Matrizen, gibt es verschiedene Abspeicherungs-Formate. Die wohl bekanntesten sind das CRS- (Compressed Row Storage) und das CCS-Format (Compressed Column Storage). Die beiden Formate sind von der Struktur her ähnlich aufgebaut. Da sich für Matrix-Vektor-Multiplikationen, die bei Eigenwert-Verfahren häufig verwendet werden, das CRS-Format besser eignet, wird dieses verwendet und nun im Detail vorgestellt. Noch ausführlicher wird es in [1] diskutiert.

Das CRS-Format verwendet folgende Datenstruktur:

Datenstruktur 1 *Sei A eine $m \times n$-Matrix mit e Nicht-Null Einträgen, dann wird diese im CRS-Format folgendermaßen abgelegt:*

```
A = record
  row_ptr   :   array [1..m+1] of INTEGER
  value     :   array [1..e] of REAL
  col_ind   :   array [1..e] of INTEGER
end record
```

Abbildung 5.1: Compressed Row Storage-Format

Das Feld row_ptr spezifiziert, an welchen Stellen in den Vektoren value und col_ind die Zeilen beginnen beziehungsweise enden. Der Vektor value enthält die k Nicht-Null-Einträge der Matrix. Im Feld col_ind wird für jeden Nicht-Null-Eintrag beschrieben, in welcher Spalte dieser sich befindet.

Das folgende Beispiel verdeutlicht das Format:

Beispiel 1 *Sei*

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

dann ergibt sich folgende Speicherung im CRS-Format:

row_ptr:	*1*	*3*	*4*	*5*	*9*			
value:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
col_ind:	*1*	*3*	*2*	*3*	*1*	*2*	*3*	*4*

5.2 Matrix-Vektor-Produkt für dünnbesetzte Matrizen

Eine zentrale Operation des Jacobi-Davidson-Verfahrens ist die Matrix-Vektor-Multiplikation.

Algorithmus 1 *Seien $A \in \mathbb{R}^{n \times n}$ und $x \in \mathbb{R}^n$, dann wird eine Matrix-Vektor-Multiplikation $y = Ax$ im CRS-Format folgendermaßen realisiert:*

```
for i = 1, ..., n do
    y(i) = 0
    for j = row_ptr(i), ..., row_ptr(i+1) - 1 do
        y(i) = y(i) + value(j) * x(col_ind(j))
    end for
end for
```

Abbildung 5.2: Matrix-Vektor-Produkt im CRS-Format

Beispiel 2 *Gegeben sei die Matrix*

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

aus dem vorherigen Beispiel im CRS-Format, dann ergibt sich für $y = Ax$ mit $x \in \mathbb{R}^4$:

$$y = \begin{pmatrix} 1 \cdot x_1 + 2 \cdot x_3 \\ 3 \cdot x_2 \\ 4 \cdot x_3 \\ 5 \cdot x_1 + 6 \cdot x_2 + 7 \cdot x_3 + 8 \cdot x_4 \end{pmatrix}$$

5.3 Aufteilung von dünnbesetzten Matrizen

Da das Jacobi-Davidson Programm für Parallelrechner implementiert wurde, war es nötig, eine geeignete Aufteilung der Matrix auf die Prozessoren zu finden. Im Programm wurde eine Aufteilung realisiert, bei der jeder Prozessor eine bestimmte Anzahl Zeilen der Matrix A und die dazu entsprechenden Komponenten der rechten Seite b erhält. Es wird versucht die Zeilen so aufzuteilen,

dass auf jedem Prozessor annähernd gleich viele Rechenoperationen durchgeführt werden. Diese Aufteilung wird im Folgenden detailliert erläutert.

Sei A eine Matrix mit e Nicht-Null-Elementen, die auf p Prozessoren $k = 0, ..., p-1$ aufgeteilt werden soll, dann werden jedem Prozessor n_k aufeinander folgende Zeilen mit insgesamt e_k Nicht-Null-Elementen zugeteilt, so dass gilt:

$$n = \sum_{k=0}^{p-1} n_k \text{ und } e = \sum_{k=0}^{p-1} e_k$$

Die erste Zeile g_k, die dem Prozessor k zugewiesen wird ist damit gegeben durch:

$$g_k = \sum_{i=0}^{k-1} n_i + 1$$

Sei nun z_i die Anzahl der Nicht-Null-Elemente der i-ten Zeile von A, dann werden dem Prozessor k

$$e_k(g_k, n_k) = \sum_{i=g_k}^{g_k+n_k-1} z_i$$

Nicht-Null-Elemente zugewiesen.

Die Rechenoperationen sollen gleichmäßig auf die Prozessoren verteilt werden, daher ist es notwendig die Komplexität der einzelnen Operationen zu betrachten. Da bei größeren Matrizen die skalaren Operationen zu vernachlässigen sind, werden im Folgenden nur die Matrix-Vektor- und die Vektor-Vektor-Operationen betrachtet. Die Anzahl der Operationen einer Matrix-Vektor-Multiplikation ist proportional zur Anzahl der besetzten Elemente e. Bei den Vektor-Vektor-Operationen verhält sich die Anzahl der durchzuführenden Operationen proportional zur Anzahl der Zeilen n der Systemmatrix. Damit ergibt sich für den Prozessor k, bei s Matrix-Vektor-Operationen pro Iteration, ein Anteil von

$$\frac{se_k + \xi n_k}{se + \xi n} \text{ mit } \xi \in \mathbb{R}$$

am Gesamtaufwand. Der Parameter ξ stellt gleichzeitig ein Maß für die Anzahl der Vektor-Vektor-Operationen pro Iteration, sowie für die Ausführungszeiten von arithmetischen, logischen und Speicher-Operationen dar. Ziel ist es nun die Parameter ξ und $n_k, k = 1, ..., n$ so zu wählen, dass das obige Verhältnis für alle Prozessoren ungefähr gleich groß wird. Es lassen sich durch geeignete Wahl von ξ zwei Extremfälle konstruieren:

- $\xi \to \infty$: jeder Prozessor erhält gleich viele Zeilen
- $\xi = 0$: jeder Prozessor erhält gleich viele Nicht-Null-Elemente

Die Rechenlast ist gleichmäßig auf die einzelnen Prozessoren verteilt, wenn jeder Prozessor den p-ten Teil aller Operationen durchführt. Deshalb ergibt sich nach [1] folgendes Kriterium für die Verteilung der Zeilen:

$$n_k = \begin{cases} \min\limits_{1 \le t \le n-g_k+1} \left\{ t \,\middle|\, \frac{se_k(t)+\xi t}{se+\xi n} \ge \frac{1}{p} \right\} & \text{für } k = 0, ..., q \\ n - \sum_{i=0}^{q} n_i & \text{für } k = q+1 \\ 0 & \text{sonst} \end{cases}$$

Für die ersten Prozessoren 0 bis q kann die Ungleichung erfüllt werden. Die restlichen Zeilen erhält der Prozessor $q + 1$. Bei kleineren Matrizen kann es passieren, dass die letzten Prozessoren keine Daten erhalten. Bei größeren Matrizen gilt für gewöhnlich $q = p - 1$ (alle Daten werden mit Hilfe der Ungleichung verteilt) oder $q + 1 = p - 1$ (ein Prozessor erhält die restlichen Zeilen).

Eine Fortführung des ersten Beispiels verdeutlicht die Aufteilung:

Beispiel 3 *Die Matrix*

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

wurde folgendermaßen im CRS-Format abgespeichert:

row_ptr:	*1*	*3*	*4*	*5*	*9*			
value:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
col_ind:	*1*	*3*	*2*	*3*	*1*	*2*	*3*	*4*

Diese soll nun auf 2 Prozessoren aufgeteilt werden. Es wird jeweils angegeben, welche Inhalte auf welchem Prozessor gespeichert werden. Zunächst wird der Grenzfall $\xi \to \infty$ („gleich viele Zeilen“) betrachtet:

Prozessor 0:	*1*	*2*	*3*		
Prozessor 1:	*4*	*5*	*6*	*7*	*8*

Für $\xi = 0$ („gleich viele Nicht-Null-Elemente“) ergibt sich folgende Aufteilung:

Prozessor 0:	*1*	*2*	*3*	*4*
Prozessor 1:	*5*	*6*	*7*	*8*

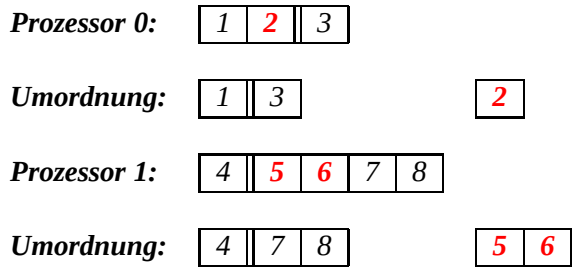
Bei einem Matrix-Vektor-Produkt muss nun jeder Prozessor Koeffizienten seiner Zeilen mit lokal gespeicherten Vektor-Komponenten, aber auch mit nicht-lokalen Vektor-Komponenten multiplizieren. Daher werden die Matrixeinträge auf den einzelnen Prozessoren in Blöcken angeordnet, so dass in Block i nur Einträge stehen, die mit Vektor-Komponenten des Prozessors i multipliziert werden. Der Block k ist dabei der Block, der auf Prozessor k zu lokalen Zugriffen führt und wird als Erstes abgelegt. Innerhalb eines Blocks werden die Daten zeilenweise, nach Spaltenindex aufsteigend angeordnet.

Das Beispiel wird wieder fortgeführt um die Aufteilung zu verdeutlichen:

Beispiel 4 *Gegeben sei die Matrix*

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

aus dem vorherigen Beispiel nach Parameter $\xi = \infty$ verteilt, dann wird diese folgendermaßen auf den Prozessoren umgeordnet, um die lokalen und nicht-lokalen Blöcke zu trennen.



Natürlich benötigt man jetzt eine neue Datenstruktur, um die einzelnen Blöcke zu speichern.

Datenstruktur 2 *Die Matrix A wird als verteilte Datenstruktur folgendermaßen in Blöcken abgelegt:*

```
A = record
  block_id   :  array [1..b_k] of INTEGER
  row_ptr    :  array [1..n_k+1,1..b_k] of INTEGER
  value      :  array [1..e_k] of REAL
  col_ind    :  array [1..e_k] of INTEGER
end record
```

Abbildung 5.3: CRS-Format für verteilte Matrizen

Im Feld block_id sind die Nummern der Blöcke gespeichert, wobei gilt block_id(1) = k. Die Einträge des Vektors row_ptr spezifizieren jeweils den Zeilen-Beginn in den Blöcken. Die Felder value und col_ind enthalten wie bisher die Inhalte der Matrix und die entsprechenden Spaltenindizes.

5.4 Matrix-Vektor-Produkt für verteilte dünnbesetzte Matrizen

Nun lässt sich ein Matrix-Vektor-Produkt pro Prozessor wie folgt implementieren:

Algorithmus 2 *Eine Matrix-Vektor-Multiplikation $y = Ax$ wird beim Block-weise-verteilten CRS-Format realisiert als Summe eines lokalen und eines nicht-lokalen Matrix-Vektor-Produktes. Auf jedem Prozessor werden die folgenden Operationen durchgeführt:*

```
lokal:
for i = 1, ..., n_p do
    y(i) = 0
    for j = row_ptr(i, 1), ..., row_ptr(i+1, 1) - 1 do
        y(i) = y(i) + value(j) * x(col_ind(j)-beginn-1)
    end for
end for

nicht-lokal:
verteile Komponenten der rechten Seite
for k = 1, ..., p do
    for i = 1, ..., n_p do
        for j = row_ptr(i, k), ..., row_ptr(i+1, k) - 1 do
            y(i) = y(i) + value(j) * x(col_ind(j)-beginn-1)
        end for
    end for
end for
summiere Komponenten des Lösungsvektors
```

Abbildung 5.4: Matrix-Vektor-Produkt für verteilte Matrizen im CRS-Format

Dabei bezeichnet beginn den Index der ersten Zeile des aktuellen Prozessors.

5.5 Speicherung von Bandmatrizen

In den folgenden Kapiteln wird genauer auf die Speicherung von Bandmatrizen eingegangen. Bandmatrizen können auf viele unterschiedliche Arten abgespeichert werden. Im Programm wurde die folgende gewählt:

Datenstruktur 3 *Sei A eine $m \times n$-Bandmatrix mit n_l Sub- und n_r Superdiagonalen, dann wird diese wie folgt zeilenweise abgespeichert:*

```
A = record
  value  :  array [1 .. (n_l + 1 + n_r) · m] of REAL
  n_l    :  INTEGER
  n_r    :  INTEGER
end record
```

Abbildung 5.5: Band-Matrix-Format

Dabei enthält value die Inhalte der Matrix und n_l und n_r spezifizieren die Anzahl der Sub- beziehungsweise Superdiagonalen.

Es ist zu beachten, dass in den ersten und letzten Zeilen der Matrix nicht alle Nebendiagonalen vorhanden sind. Eine entsprechende Speicher-optimierende Datenhaltung führt aber zu einem komplizierterem, deutlich langsameren Algorithmus. Daher werden diese Elemente als Dummy-Elemente mit geführt. Damit ergibt sich eine Speicherung, in der pro Zeile zunächst n_l Subdiagonal- dann das Diagonal- und dahinter n_r Superdiagonalelemente gespeichert werden.

Das folgende Beispiel verdeutlicht die Speicherung:

Beispiel 5 *Sei*

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 6 & 7 & 8 & 9 \\ 0 & 10 & 11 & 12 \end{pmatrix}$$

dann ergibt sich als Speicherung:

value:	*0*	*0*	*1*	*2*	*0*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*0*

n_l: *2*
n_r: *1*

5.6 Matrix-Vektor-Produkt für Bandmatrizen

In dieser Darstellung lässt sich sehr leicht eine effiziente Matrix-Vektor-Multiplikation implementieren.

Algorithmus 3 *Seien $A \in \mathbb{R}^{n\times n}$ und $x \in \mathbb{R}^n$, dann wird eine Matrix-Vektor-Multiplikation $y = Ax$ im Bandmatrizen-Format folgendermaßen realisiert:*

```
k = 1
for i = 1, ..., n do
    y(i) = dot_product(value(k:k+n_l+n_r), x(i:i+n_l+n_r))
    k = k + n_l + n_r + 1
end for
```

Abbildung 5.6: Matrix-Vektor-Produkt im Band-Format

Es wird hier vorausgesetzt, dass auch der Vektor entsprechende Dummy-Elemente beinhaltet, dieses ist aber bei dem im Kapitel 5.8 vorgestellten Produkt für verteilte Matrizen, welches auch im Programm verwendet wird, automatisch gegeben. Ein Beispiel verdeutlicht den Algorithmus:

Beispiel 6 *Gegeben seien die Matrix A aus dem vorherigen Beispiel*

$A =$

0	*0*	*1*	*2*	*0*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*0*

im Bandformat gespeichert und ein Vektor $x \in \mathbb{R}^4$. Zur Berechnung des Matrix-Vektorproduktes $y = Ax$ wird der Vektor

$$vec = \begin{pmatrix} 0 & 0 & x_1 & x_2 & x_3 & x_4 & 0 \end{pmatrix}^T$$

erzeugt. y ergibt sich nach dem Algorithmus als:

$$y = \begin{pmatrix} 0 \cdot vec(1) + 0 \cdot vec(2) + 1 \cdot vec(3) + 2 \cdot vec(4) \\ 0 \cdot vec(2) + 3 \cdot vec(3) + 4 \cdot vec(4) + 5 \cdot vec(5) \\ 6 \cdot vec(3) + 7 \cdot vec(4) + 8 \cdot vec(5) + 9 \cdot vec(6) \\ 10 \cdot vec(4) + 11 \cdot vec(5) + 12 \cdot vec(6) + 0 \cdot vec(7) \end{pmatrix}$$

$$= \begin{pmatrix} 1 \cdot x_1 + 2 \cdot x_2 \\ 3 \cdot x_1 + 4 \cdot x_2 + 5 \cdot x_3 \\ 6 \cdot x_1 + 7 \cdot x_2 + 8 \cdot x_3 + 9 \cdot x_4 \\ 10 \cdot x_2 + 11 \cdot x_3 + 12 \cdot x_4 \end{pmatrix}$$

5.7 Aufteilung von Bandmatrizen

Bei der beschriebenen Speicherung der Bandmatrizen bietet es sich wieder an, die Matrix zeilenweise auf die einzelnen Prozessoren zu verteilen. Da es sich um Bandmatrizen handelt und somit jede Zeile gleich viele Einträge enthält, tritt hier auch nicht das Problem auf, die Operationen gleichmäßig verteilen zu müssen. Jeder Prozessor erhält zusätzlich die entsprechenden Komponenten der rechten Seite.

Beispiel 7 *Folgende Speicherung ergab sich für die Matrix A:*

value:	*0*	*0*	*1*	*2*	*0*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*0*

Nach einer Aufteilung auf 2 Prozessoren ergibt sich:

Prozessor 0:	*0*	*0*	*1*	*2*	*0*	*3*	*4*	*5*

Prozessor 1:	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*0*

Es lassen sich hier verschiedene Ansätze bei der Wahl der Blockgröße wählen. Der intuitivste Ansatz ist es, als Blockgröße den abgerundeten Quotienten aus der Anzahl der Zeilen der Matrix und der Anzahl der eingesetzten Prozessoren zu wählen.

Beispiel 8 *Sei A eine 9×9-Matrix, die auf 4 Prozessoren bearbeitet werden soll, dann ergibt sich als Blockgröße NB:*

$$NB = \left\lfloor \frac{9}{4} \right\rfloor = 2$$

Damit erhalten die ersten drei Prozessoren 2 Zeilen und der letzte Prozessor 3 Zeilen.

Dass diese Strategie zur Wahl der Blockgröße nicht immer optimal ist, zeigt das nächste Beispiel:

Beispiel 9 *Sei A eine 9×9-Matrix, die auf 5 Prozessoren bearbeitet werden soll, dann ergibt sich als Blockgröße NB:*

$$NB = \left\lfloor \frac{9}{5} \right\rfloor = 1$$

Damit erhalten die ersten vier Prozessoren eine Zeile und der letzte Prozessor 4 Zeilen.

Es lässt sich deutlich erkennen, dass der letzte Prozessor gegenüber den anderen völlig überlastet ist, wodurch die anderen Prozessoren große Leerlaufzeiten haben. Weiterhin ist in beiden Aufteilungen die Anzahl der Zeilen des letzten Prozessors größer als die Blockgröße. Dies ist aber für „ScaLAPACK“-Routinen [16], die zur Implementierung eines Bandlösers verwendet wurden (siehe Kapitel 7.6), unzulässig.

Eine bessere Aufteilung, bei der die Zeilen gleichmäßiger aufgeteilt werden und der letzte Prozessor höchstens so viele Zeilen erhält, wie die Blockgröße angibt, erhält man durch Aufrunden des Quotienten. Betrachtet man nochmals das letzte Beispiel, so erkennt man, dass die Aufteilung wesentlich gleichmäßiger aussieht.

Beispiel 10 *Sei A eine 9×9-Matrix, die auf 5 Prozessoren bearbeitet werden soll, dann ergibt sich als Blockgröße NB:*

$$NB = \left\lfloor \frac{9}{5} + 1 - \varepsilon \right\rfloor = 2$$

Damit erhalten die ersten vier Prozessoren zwei Zeilen und der letzte Prozessor eine Zeile.

Die maximale Anzahl an Zeilen je Prozessor konnte von 4 auf 2 gesenkt werden. Es gibt allerdings auch Fälle, in denen die Aufteilung auf den ersten Blick ungleichmäßiger als beim Abrunden zu sein scheint.

Beispiel 11 *Sei A eine 9×9-Matrix, die auf 4 Prozessoren bearbeitet werden soll, dann ergibt sich als Blockgröße NB:*

$$NB = \left\lfloor \frac{9}{4} + 1 - \varepsilon \right\rfloor = 3$$

Damit erhalten die ersten drei Prozessoren 3 Zeilen und der letzte Prozessor keine Zeile.

Beim Abrunden erhielten die ersten drei Prozessoren nur 2 Zeilen und der letzte Prozessor drei Zeilen. Betrachtet man dies genauer, so ist die neue Aufteilung keine wirkliche Verschlechterung, da die maximale Anzahl an Zeilen pro Prozessor bei 3 geblieben ist. Insgesamt betrachtet bedeutet dies also keinen langsameren Programmablauf. Der Unterschied ist nur, dass bei dieser Aufteilung Prozessor 3 längere Wartezeiten hat, während zuvor die ersten 3 Prozessoren auf den dritten Prozessor warten mussten. Diese Aussage lässt sich allgemein beweisen.

Satz 1 *Verwendet man als Blockgröße den aufgerundeten Quotienten aus der Anzahl der Zeilen der Systemmatrix n und der Anzahl der eingesetzten Prozessoren p_{ges}, so ist die Anzahl der Zeilen je Prozessor immer kleiner oder gleich der Anzahl der Zeilen je Prozessor beim Abrunden des Quotienten.*

Beweis:
1. p_{ges} teilt n:
$\Rightarrow \left\lceil \frac{n}{p_{ges}} \right\rceil = \left\lceil \frac{n}{p_{ges}} + 1 - \varepsilon \right\rceil \Rightarrow$ *Die Blockgrößen sind identisch und jeder Prozessor erhält in beiden Aufteilungen so viele Zeilen, wie die Blockgröße angibt.*

2. p_{ges} teilt n nicht:
In der Aufteilung durch Abrunden erhält der letzte Prozessor in diesem Fall immer die meisten Zeilen und zwar einen Block der Blockgröße und den nicht aufgeteilten Rest. Die maximale Anzahl an Zeilen pro Prozessor n_{ab} beträgt also:

$$n_{ab} = \left\lceil \frac{n}{p_{ges}} \right\rceil + n - p_{ges} \cdot \left\lceil \frac{n}{p_{ges}} \right\rceil$$

Bei der Aufteilung durch Aufrunden ist die maximale Anzahl an Zeilen je Prozessor n_{auf} beschränkt durch die Blockgröße. Diese lässt sich im gegebenen Fall ($p_{ges} \nmid n$) einfacher darstellen:

$$n_{auf} = \left\lceil \frac{n}{p_{ges}} \right\rceil + 1$$

Damit ergibt sich:

$$n_{ab} - n_{auf} = \underbrace{n - p_{ges} \cdot \left\lceil \frac{n}{p_{ges}} \right\rceil}_{\geq 1, da\ p_{ges} \nmid n} - 1 \geq 1$$

Da sich im Allgemeinen durch Aufrunden des Quotienten eine gleichmäßigere Aufteilung ergibt, die nie schlechter ist, als die durch Abrunden und zusätzlich damit gewährleistet ist, dass die Anzahl der Zeilen des letzten Prozessors nie größer ist als die Blockgröße ist („ScaLAPACK“-Bedingung), wird im Programm als Blockgröße der aufgerundete Quotient verwendet. Eine Strategie, die im Allgemeinen die beste Aufteilung erzeugt, ist es, zunächst abzurunden und dann jeweils die Zeilen des letzten Prozessors, die übrig bleiben, auf die ersten Prozessoren zu verteilen. Allerdings werden nicht die letzten Zeilen verteilt, sondern die ersten Prozessoren erhalten immer noch zusammenhängende Blöcke, allerdings mit einer Zeile mehr. Diese Aufteilung wurde nicht verwendet, da diese sich nicht in „ScaLAPACK“ [16] realisieren lässt, da dort immer mit einer festen Blockgröße und zyklischer Verteilung gearbeitet wird.

5.8 Matrix-Vektor-Produkt für verteilte Bandmatrizen

Auch beim verteilten Matrix-Vektor-Produkt für Bandmatrizen tritt wieder das Problem auf, dass bei einer Matrix-Vektor-Multiplikation lokale Matrix-Einträge mit nicht-lokalen Vektor-Einträgen multipliziert werden müssen. Es lässt sich aber zeigen, dass bei geeigneter Wahl der Anzahl der

Prozessoren in Abhängigkeit von der Anzahl der Zeilen und der Bandbreite immer nur eine Kommunikation mit den benachbarten Prozessoren stattfindet.

Satz 2 *Verteilt man eine $n \times n$-Bandmatrix mit n_l Subdiagonalen und n_r Superdiagonalen auf p_{ges} Prozessoren, so findet beim Matrix-Vektor-Produkt nur Kommunikation mit den Nachbarprozessoren statt, falls gilt:*

$$\left(p_{ges} \mid n \wedge p_{ges} \leq \frac{n}{max(n_l, n_r)}\right) \vee \left(p_{ges} \nmid n \wedge p_{ges} \leq \frac{n}{max(n_l, n_r)+1}\right)$$

Beweis:
Sei A eine wie beschrieben abgespeicherte $n \times n$-Bandmatrix mit n_l Sub- und n_r Superdiagonalen, x ein n-dimensionaler Vektor und $y = Ax$ das zu berechnende Matrix-Vektor-Produkt. Für die i-te Komponente von y ergibt sich:

$$y_i = \sum_{j=max(1,i-n_l)}^{min(n,i+n_r)} a_{ij} \cdot b_j$$

das heißt zur Berechnung von y_i werden nur die Komponenten $b_j, j = i-n_l, ..., i+n_r$ benötigt. Sei nun o.B.d.A. p derjenige Prozessor, auf dem die i-te Zeile der Matrix gespeichert ist. Dann reicht es zu zeigen, dass auf den Prozessoren $p-1$ bis $p+1$ die Zeilen $i-n_l$ bis $i+n_r$ gespeichert sind.

Betrachtet man dies genauer, dann reicht es zu zeigen, dass die Zeile $i-n_l$ auf Prozessor $p-1$ oder p und die Zeile $i+n_r$ auf Prozessor $p+1$ oder p abgelegt wurden.

Sei nun i die erste Zeile des Prozessors p, dann müssen damit n_l Zeilen auf Prozessor $p-1$ abgelegt sein. Ist i die letzte Zeile des Prozessors p, dann müssen n_r Zeilen auf Prozessor $p+1$ abgelegt sein. In allen anderen Fällen werden weniger nicht-lokale Vektor-Komponenten benötigt.

Jeder Prozessor (außer dem letzten) erhält $\left[\frac{n}{p_{ges}} + 1 - \varepsilon\right]$ Zeilen. Nach obiger Betrachtung müssen auf jedem Prozessor mindestens $max(n_l, n_r)$ Zeilen abgelegt werden. Es ergibt sich:

$$\left[\frac{n}{p_{ges}} + 1 - \varepsilon\right] \geq max(n_l, n_r)$$

Der letzte Prozessor, dem noch Zeilen zugeteilt wurden, stellt diese dem vorherigen zur Verfügung. Da diese ohnehin die letzten Zeilen der Matrix sind ist es auch kein Problem, wenn auf diesem weniger als n_r Zeilen gespeichert sind. Die nachfolgenden Prozessoren erhalten keine Zeilen und stellen somit kein Problem dar.

Die hergeleitete Ungleichung lässt sich durch eine Fallunterscheidung noch weiter vereinfachen:

$$p_{ges} \mid n : \frac{n}{p_{ges}} \geq max(n_l, n_r) \Leftrightarrow p_{ges} \leq \frac{n}{max(n_l, n_r)}$$
$$p_{ges} \nmid n : \frac{n}{p_{ges}} + 1 \geq max(n_l, n_r) \Leftrightarrow p_{ges} \leq \frac{n}{max(n_l, n_r)+1}$$

Die folgenden Beispiele verdeutlichen den hergeleiteten Zusammenhang:

Beispiel 12 *Seien*

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 6 & 7 & 8 & 9 \\ 0 & 10 & 11 & 12 \end{pmatrix}, \; x = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

dann ergibt sich als Aufteilung auf zwei Prozessoren:

Prozessor 0:	$A =$	0	0	1	2	0	3	4	5	$x =$	1	2
Prozessor 1:	$A =$	6	7	8	9	10	11	12	0	$x =$	3	4

Es ergibt sich folgendes Matrix-Vektorprodukt:

$$y = Ax = \begin{pmatrix} 1\cdot 1 & + & 2\cdot 2 & & & & \\ 3\cdot 1 & + & 4\cdot 2 & + & 5\cdot \mathbf{3} & & \\ 6\cdot \mathbf{1} & + & 7\cdot \mathbf{2} & + & 8\cdot 3 & + & 9\cdot 4 \\ & & 10\cdot \mathbf{2} & + & 11\cdot 3 & + & 12\cdot 4 \end{pmatrix}$$

Die fettgedruckten (blauen) Vektor-Werte sind nicht-lokale Komponenten und werden vom Nachbarprozessor benötigt. Es gilt hier:

$$p_{ges} = 2 \leq \frac{4}{2} = \frac{n}{max(n_l, n_r)}$$

Also findet hier nur Kommunikation mit den direkten Nachbar-Prozessoren statt.

Beispiel 13 *Für A, b aus dem vorherigen Beispiel ergibt sich als Aufteilung auf vier Prozessoren:*

Prozessor 0:	$A =$	0	0	1	2	$x =$ 1
Prozessor 1:	$A =$	0	3	4	5	$x =$ 2
Prozessor 2:	$A =$	6	7	8	9	$x =$ 3
Prozessor 3:	$A =$	10	11	12	0	$x =$ 4

Es ergibt sich folgendes Matrix-Vektorprodukt:

$$y = Ax = \begin{pmatrix} 1\cdot 1 & + & 2\cdot \mathbf{2} & & & & \\ 3\cdot \mathbf{1} & + & 4\cdot 2 & + & 5\cdot \mathbf{3} & & \\ 6\cdot \underline{\mathbf{1}} & + & 7\cdot \mathbf{2} & + & 8\cdot 3 & + & 9\cdot \mathbf{4} \\ & & 10\cdot \underline{\mathbf{2}} & + & 11\cdot \mathbf{3} & + & 12\cdot 4 \end{pmatrix}$$

Die fettgedruckten (blauen) Vektor-Werte sind nicht-lokale Komponenten, die auf den Nachbarprozessor vorliegen. Die unterstrichenen (roten) Werte, liegen nicht auf Nachbarprozessoren vor. Dies lässt sich auch daran erkennen, dass das Kriterium für benachbarte Kommunikation nicht erfüllt ist.

$$p_{ges} = 4 \leq \frac{4}{2} = \frac{n}{max(n_l, n_r)} \lightning$$

Durch die sehr eingeschränkte Kommunikation lässt sich eine Matrix-Vektor-Multiplikation hier leicht umsetzen.

Algorithmus 4 *Eine Matrix-Vektor-Multiplikation im Bandmatrizen-Format wird bei einer wie beschrieben verteilten Bandmatrix folgendermaßen realisiert. Dabei werden die Operationen auf jedem Prozessor durchgeführt.*

```
lade n_l Vektorkomponenten des vorherigen Prozessors in vec(1:n_l)
schreibe eigene Vektorkomponenten in vec(n_l+1:n_l+nnp)
lade n_r Vektorkomponenten des nächsten Prozessors in
   vec(n_l+nnp+1:n_l+nnp+n_r)
k = 1
for i = 1, ..., n_p do
    y(i) = dot_product(value(k:k+n_l+n_r), vec(i:i+n_l+n_r))
    k = k + n_l + n_r + 1
end for
```

Abbildung 5.7: Matrix-Vektor-Produkt für verteilte Matrizen im CRS-Format

Das folgende Beispiel verdeutlicht die Berechnung des Produktes:

Beispiel 14 *Seien*

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 5 \\ 0 & 6 & 7 \end{pmatrix}, x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Damit ergibt sich folgende Aufteilung auf drei Prozessoren:

Prozessor 0: $A =$ | *0* | *1* | *2* | $x =$ | *1* |

Prozessor 1: $A =$ | *3* | *4* | *5* | $x =$ | *2* |

Prozessor 2: $A =$ | *6* | *7* | *0* | $x =$ | *3* |

Jeder Prozessor lädt nun die Vektor-Komponenten seiner Nachbarprozessoren und erzeugt so einen Vektor vec, in dessen ersten n_l Komponenten, die letzten Komponenten des Prozessors p-1 stehen. Danach folgen die Komponenten des Prozessors selbst. Dahinter stehen die ersten n_r Komponenten des Prozessor p+1.

Prozessor 0: *vec =* | *0* | *1* | *2* |

Prozessor 1: *vec =* | *1* | *2* | *3* |

Prozessor 2: *vec =* | *2* | *3* | *0* |

Nun lässt sich in diesem Fall das Matrix-Vektorprodukt $y = Ax$ auf jedem Prozessor darstellen durch das Skalarprodukt des Zeilenvektors von A pro Prozessor mit dem jeweiligen Vektor vec.

Im Allgemeinen wird pro Komponente von y ein Skalarprodukt berechnet, in dem eine Zeile von A und ein Teil des Vektors vec eingeht.

5.9 Matrix-Matrix-Produkt

Im Jacobi-Davidson-Algorithmus tritt häufig der Fall auf, dass Produkte der Systemmatrix mit allen Iterationsvektoren berechnet werden müssen. Es muss also n-fach die Routine des Matrix-Vektor-Produktes gestartet werden.

Betrachtet man noch einmal genauer, was innerhalb der Routine abläuft so erkennt man, dass nacheinander einzelne Zeilen mit den Vektoren multipliziert werden. Da der Cache eines Rechners nur sehr beschränkt, die Zeilen aber in der Regel sehr lang sind, werden die Zeilen also jeweils eingelagert und anschließend wieder ausgelagert und durch die nächste Zeile ersetzt. Diese sehr langsamen

Ein/Auslagerungsprozesse müssen also n-fach durchgeführt werden.

Ein besserer Ansatz ist deshalb, die n Matrix-Vektor-Produkte als ein Matrix-Matrix-Produkt aufzufassen. Für den oben erwähnten Fall der Iterationsvektoren bildet das noch nicht einmal zusätzlichen Aufwand, da diese ohnehin am leichtesten in einer Spalten-Matrix abgelegt werden können. Verwendet man diesen Ansatz, so muss jede Zeile nur einmal geladen werden und kann direkt mit den entsprechenden Teilen aller Vektoren multipliziert werden. Der Algorithmus des Matrix-Matrix-Produktes ist für jedes Matrix-Format nahezu identisch mit dem des bereits bekannten Matrix-Vektor-Produktes. Der einzige Unterschied ist, dass die Operationen für alle Vektoren, anstatt nur für eine wie beim Matrix-Vektor-Produkt, durchgeführt werden müssen. Als Beispiel einer Implementierung eines Matrix-Matrix-Produktes ist hier die für Bandmatrizen angegeben.

Algorithmus 5 *Eine Matrix-Matrix-Multiplikation wird im Bandmatrizen-Format folgendermaßen realisiert.*

```
k = 1
do i = 1, n_p
    y(i,:) = matmul(value(k:k+n_l+n_r), mat(i:i+n_l+n_r,:))
    k = k + n_l + n_r + 1
end do
```

Abbildung 5.8: Matrix-Matrix-Produkt im Band-Format

Dabei stellt mat nun wieder eine Matrix dar, die die entsprechenden Komponenten des eigenen und der Nachbarprozessoren enthält.

Bei diesem zunächst trivialen Algorithmus gibt es aber eventuell ein Problem, die Matrix mat. Zur Generierung der Matrix mat müssen die Prozessoren Blöcke ihrer Matrizen austauschen. Die Sende- und Empfangs-Befehle von MPI (Message Passing Interface) [14] stellen aber zunächst nur eine Möglichkeit bereit, um Inhalte zusammenhängender Speicherbereiche zu versenden. Bei Programmiersprachen, die eine spaltenweise Speicherung verwenden, gibt es damit Probleme. Das folgende Beispiel verdeutlicht das Problem.

Beispiel 15 *Seien*

$$x_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix},\ x_2 = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix},\ x_3 = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}$$

Die Vektoren werden nun zusammengefasst zur Matrix

$$X = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

Angenommen, es würde mit zwei Prozessoren gearbeitet und Prozessor 0 wären die ersten beiden Zeilen zugeteilt und Prozessor 1 die letzte Zeile, dann muss Prozessor 0 für das Matrix-Matrix-Produkt den Block

1	*4*	*7*
2	*5*	*8*

an Prozessor 1 senden. Prozessor 1 muss den Block

3	*6*	*9*

an Prozessor 0 senden.

Es lässt sich nun direkt erkennen, dass die Koeffizienten der Blöcke bei einer spaltenweise abgespeicherten Matrix keinen zusammenhängenden Speicherbereich bilden. Der nahe liegendste Ansatz an dieser Stelle ist wohl, den Block der transponierten Matrix zu senden, da dieser dann im Speicher

zusammenhängend ist. Leider stellt das Transponieren einer Matrix eine Operation der Komplexität n^2 dar und ist somit viel zu aufwändig, um bei jedem Matrix-Matrix-Produkt durchgeführt werden zu können. An späterer Stelle wird noch ein weiterer Grund diskutiert, warum Transponieren inakzeptabel ist.

Einen besseren Ansatz bildet hier die Verwendung von abgeleiteten MPI-Datentypen [14], mit deren Hilfe man sich einen Datentyp definieren kann, der es erlaubt Blöcke von Matrizen zu senden.

Datenstruktur 4 *Zum Austausch von Blöcken bei einer zeilenweise verteilten Matrix wird folgende Datendefinition verwendet. Y ist dabei eine Matrix, an die ein Block mit anz_ze Zeilen und anz_sp Spalten aus der Matrix X gesendet werden sollen.*

```
call MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION,
        sizeofdp, ierror)
call MPI_TYPE_VECTOR(anz_ze, 1, 1,
        MPI_DOUBLE_PRECISION, row, ierror)

call MPI_TYPE_HVECTOR(anz_sp, 1,
        size(X,1) * sizeofdp, row, blsend, ierror)
call MPI_TYPE_COMMIT(blsend, ierror)

call MPI_TYPE_HVECTOR(anz_sp, 1,
        size(Y,1) * sizeofdp, row, blrecv, ierror)
call MPI_TYPE_COMMIT(blrecv, ierror)
```

Abbildung 5.9: MPI-Datendefinition zum Austausch von Blöcken

Entsprechend dieser Definition muss für jeden Block, der gesendet werden soll, ein Datentyp zum Versenden und einer zum Empfangen der Daten generiert werden.

Es wurde bisher nur das Problem bei der spaltenweisen Speicherung deutlich. Aber auch bei der zeilenweisen Speicherung kann es zu Problemen kommen. Solange alle Iterationsvektoren der Matrix verwendet werden, treten keine Probleme auf. Sind aber beispielsweise bereits einige Eigenvektoren konvergiert oder aus anderen Gründen sollen nur die ersten k Iterationsvektoren mit der Systemmatrix multipliziert werden, so liegen auch bei der zeilenweisen Speicherung keine zusammenhängenden Bereiche mehr vor. Damit lässt sich aber auch direkt erkennen, dass die oben angesprochene Variante der Multiplikation mit der transponierten Matrix nicht immer durchführbar ist. Ein kleines Beispiel verdeutlicht das Problem:

Beispiel 16 *Sei*

$$X = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

die Matrix der Iterationsvektoren x_1, x_2 und x_3. Es soll mit zwei Prozessoren gearbeitet werden und die ersten beiden Iterationsvektoren sollen mit der Systemmatrix multipliziert werden. Dann muss Prozessor 0 den Block

1	*4*
2	*5*

an Prozessor 1 senden.

Es lässt sich erkennen, dass der Block weder bei einer spaltenweisen noch bei einer zeilenweisen Speicherung im Speicher zusammenhängend ist.

5.10 Vergleich zwischen CRS- und Band-Format

In den vorherigen Kapiteln wurde ausführlich das Band-Matrix-Format beschrieben. In diesem Kapitel wird nun verdeutlicht, warum es Sinn macht, neben dem CRS-Format für willkürlich dünnbesetzte Matrizen zusätzlich das Band-Format zu verwenden, obwohl Band-Matrizen natürlich auch durch das CRS-Format dargestellt werden können.

Zunächst ist klar, dass das Band-Matrix-Format sich nur für wirkliche Band-Matrizen eignet. Das heißt eine Matrix die im inneren Bereich dünn besetzt ist, und somit nur wenige Nebendiagonalen hat, kann trotzdem zumeist besser im CRS-Format gespeichert werden. Das Band-Matrix-Format eignet sich erst dann besser, wenn die Matrix im inneren Bereich sehr dicht besetzt ist.

Um dies zu verdeutlichen, wird zunächst der Speicheraufwand in beiden Formaten betrachtet. Gegeben sei eine $n \times n$-Matrix mit e Nicht-Null-Elementen und n_l Sub- und n_r Superdiagonalen. Dann benötigt diese im CRS-Format

$$SP_{CRS} = (n + 1 + 2 \cdot e) \cdot sizeof(double)$$

Byte Speicherplatz, da jeweils die Zeilenbeginne und für jeden Koeffizienten der Spaltenindex mit abgespeichert werden muss. Im Band-Format reicht es aus, die Koeffizienten selbst abzuspeichern und es werden damit nur

$$SP_{Band} = (n \cdot (n_l + 1 + n_r)) \cdot sizeof(double)$$

Byte benötigt.

Für den Speicherbedarf ergibt sich damit folgender Satz.

Satz 3 *Gegeben sei eine $n \times n$-Matrix mit e Nicht-Null-Elementen und n_l Sub- und n_r Superdiagonalen. Diese benötigt genau dann weniger Speicherplatz im Band-Format als im CRS-Format, wenn gilt:*

$$n \cdot (n_l + 1 + n_r) < n + 1 + 2 \cdot e$$

Viel entscheidender als der Speicherbedarf ist aber noch der Rechenaufwand. Dieser ist beim CRS-Format größer, denn es wird jeweils zweifach indiziert auf die rechte Seite zugegriffen, denn zu jedem Koeffizienten existiert ja ein entsprechender Spaltenindex und bei der Multiplikation muss jeder Koeffizient mit dem dazu passenden Koeffizienten der rechten Seite multipliziert werden.

Auch hier lässt sich in etwas abstrakter Form der folgende Satz formulieren.

Satz 4 *Gegeben sei eine $n \times n$-Matrix mit e Nicht-Null-Elementen und n_l Sub- und n_r Superdiagonalen. Weiterhin sei MT die CPU-Zeit für eine Multiplikation zweier Zahlen und IT die Zeit für einen indizierten Speicherzugriff. Dann kann ein Matrix-Vektor-Produkt im Band-Format genau dann schneller durchgeführt werden als im CRS-Format, wenn gilt:*

$$(n \cdot (n_l + 1 + n_r)) \cdot (MT + IT) < (n + 1 + 2 \cdot e) \cdot (MT + 2 \cdot IT)$$

Kapitel 6

Schnittstellen

6.1 Matrix-Modul

Das Jacobi-Davidson-Verfahren bestimmt näherungsweise Eigenwerte einer beliebigen Matrix. Im Algorithmus wird die Matrix ausschließlich in Matrix-Vektor-Produkten verwendet. Für das Verfahren ist also vollkommen unerheblich, welche Besetzungsstruktur die Matrix besitzt und auf welche Art und Weise ein Matrix-Vektor-Produkt realisiert ist. Daher ist es sinnvoll, die gesamte Verwaltung der Matrix vom Algorithmus abzukoppeln, so dass dieser nur noch den Auftrag gibt, eine solche Multiplikation durchzuführen. Dies stellt einen wesentlichen Punkt bei der Reorganisation des Programms dar.

Schnittstelle 1 *im Wesentlichen sollte eine Schnittstelle zwischen Algorithmus und Matrix-Verwaltung die folgende Form besitzen:*

subroutine *mvm(y, x, ...)*

x ist der Vektor, der mit der in der Matrix-Verwaltung gespeicherten Matrix multipliziert werden soll. In y wird das Ergebnis des Matrix-Vektorproduktes abgelegt.

Die gesamte Matrix-Verwaltung erfolgt in einem Modul. Dieses stellt alle Variablen und Datenstrukturen, die die Matrix und deren Verteilung beschreiben, bereit. Weiterhin stellt das Modul Unterprogramme bereit, die den Auftrag, ein Matrix-Vektorprodukt beziehungsweise Matrix-Matrix-Produkt zu berechnen, weiterverarbeiten. In den Unterprogrammen wird je nach Besetzungsstruktur der Matrix die entsprechende Routine aufgerufen, die die Multiplikation für diese Struktur realisiert.

Schnittstelle 2 *Ein Unterprogramm, welches die Schnittstelle für das Matrix-Vektorprodukt realisiert, hat einen Aufbau der folgenden Form.*

```
subroutine mvm(y, x, ...)
    select case (matformat)

    *** CRS - Format ***
    case (1)
        call lmvmcrs(y, x, ...)
        call nlmvmcrs(y, ...)

    *** Bandmatrix - Format ***
    case (2)
        call mvmband(y, x, ...)

    ...

    end select
end subroutine mvm
```

Abbildung 6.1: Schnittstelle für das Matrix-Vektor-Produkt

Es werden verschiedene Routinen benötigt, um die Matrix einzulesen, Kommuniktationschemata zu berechnen und die Matrix auf die Prozessoren zu verteilen. Diese Unterprogramme benötigen alle Zugriff auf Variablen, die die Matrix beschreiben. Auch hier eignet sich der Einsatz eines Moduls sehr gut, da so die Schnittstellen zu den Funktionen nicht aufgebläht werden. Anstelle dessen wird den Routinen nur Zugriff auf die von ihnen benötigten Variablen der Matrix gegeben, wodurch eine Datenkapselung möglich wird. Weiterhin hat dies den großen Vorteil, dass keine Kopien von Variablen angelegt werden müssen, sondern direkt auf die eigentlichen Speicherbereiche zugegriffen werden kann.

Schnittstelle 3 *Die Variablen, die die Matrix beschreiben, können folgendermaßen einzeln für Unterprogramme freigegeben werden.*

use <Modulname>, only: <Variablenliste>

Nachdem die Matrix einmal eingelesen und auf die Prozessoren verteilt wurde, reicht die oben beschriebene Schnittstelle mvm aus, da die Matrix nur noch für Matrix-Vektor-Produkte benötigt wird.

6.2 Löser-Modul

Ein ähnliches Modul wurde auch zur Beschreibung des Gleichungssystemlösers implementiert. In diesem Modul werden alle Variablen bereit gestellt, die den Löser beschreiben beziehungsweise vom Löser verwendet werden.

Weiterhin werden auch hier zwei Schnittstellen Funktionen *simplesolv* und *complsolv* bereit gestellt, die entsprechend dem Matrix-Format einen einfachen beziehungsweise komplexeren Löser für die Korrekturgleichung starten.

Schnittstelle 4 *Die Schnittstellen-Routine simplesolv hat einen Aufbau der folgenden Form.*

```
subroutine simplesolv(k, ...)
    select case (matformat)

    *** CRS - Format ***
    case (1)
        call diagprec(eigval(k), eigvec(1, k), ...)

    *** Bandmatrix - Format ***
    case (2)
        if (use_diag_solver) then
            call diagprec(eigval(k), eigvec(1, k), ...)
            else
            call bandprec(eigval(k), eigvec(1, k), ...)
            end if
    ...

    end select
end subroutine simplesolv
```

Abbildung 6.2: Schnittelle für den einfachen Löser

Als einfacher Löser steht für das CRS-Format ein Diagonallöser und für das Bandmatrizen-Format zusätzlich ein Bandlöser zur Verfügung. Für beide Verfahren werden als komplexere Löser die CG-artigen Verfahren QMR und TFQMR angeboten. Die Möglichkeiten zur Lösung der Korrekturgleichung werden in Kapitel 7.5 genauer beschrieben.

Kapitel 7

Das Jacobi-Davidson-Programm

7.1 Überlick über die Teilaufgaben

Das Jacobi-Davidson-Programm hat einen einfachen Aufbau. Es besteht hauptsächlich aus einem Hauptprogramm und einem Hauptunterprogramm. Das Hauptprogramm steuert den Programmablauf. Von dort aus werden Routinen zum Einlesen, Aufarbeiten und Ausgeben der Daten, sowie das Hauptunterprogramm, welches den eigentlichen Jacobi-Davidson-Algorithmus realisiert, aufgerufen.

Das Hauptprogramm steuert die Durchführung der folgenden Aufgaben:

1. Einlesen der Programmparameter
2. Einlesen und Verteilen der Matrix auf die Prozessoren
3. Berechnung von Vorkonditionierern
4. Ausführung des Jacobi-Davidson-Algorithmus
5. Ausgabe der Ergebnisse

Bei dieser Aufgabenverteilung ist zu beachten, dass die Punkte 1. und 5. zweifellos völlig unabhängig von der Struktur der Matrix durchgeführt werden können. Der Jacobi-Davidson-Algorithmus ist selbstverständlich zunächst abhängig vom Aufbau der Matrix. Allerdings lässt sich bei genauerer Betrachtung feststellen, dass sich dies lediglich auf die Matrix-Vektor-Produkte bezieht.

Dies ist der Ansatzpunkt für die Modularisierung des Programms. Das Programm wurde so reorganisiert, dass dem Hauptalgorithmus die Matrix überhaupt nicht bekannt ist. Die gesamten Matrixinformationen sind nur der Routine bekannt, die das Matrix-Vektor-Produkt realisiert (siehe Kapitel 6.1). Vom Algorithmus wird lediglich diese Routine verwendet. Weiterhin wird die Lösung der Korrekturgleichung von einer abgetrennten Routine übernommen. Innerhalb der Routinen gibt es dann für jedes unterstützte Matrix-Format eine entsprechende Implementierung für das Matrix-Vektor-Produkt, beziehungsweise die Lösung der Korrekturgleichung.

7.2 Aufbau des Programmes

Im Modulplan (Abbildung 7.1) ist der Ablauf des Hauptprogramms dargestellt. Zunächst wird die Prozedur *parin* aufgerufen, die die Parameter (siehe Kapitel 8.1) des Programms einliest. Diese beschreiben im Wesentlichen das Matrix-Format, die Art der zu suchenden Eigenwerte und die Methoden, mit denen diese gesucht werden sollen. Das Unterprogramm *matopen* öffnet die binäre Matrix-Eingabedatei.

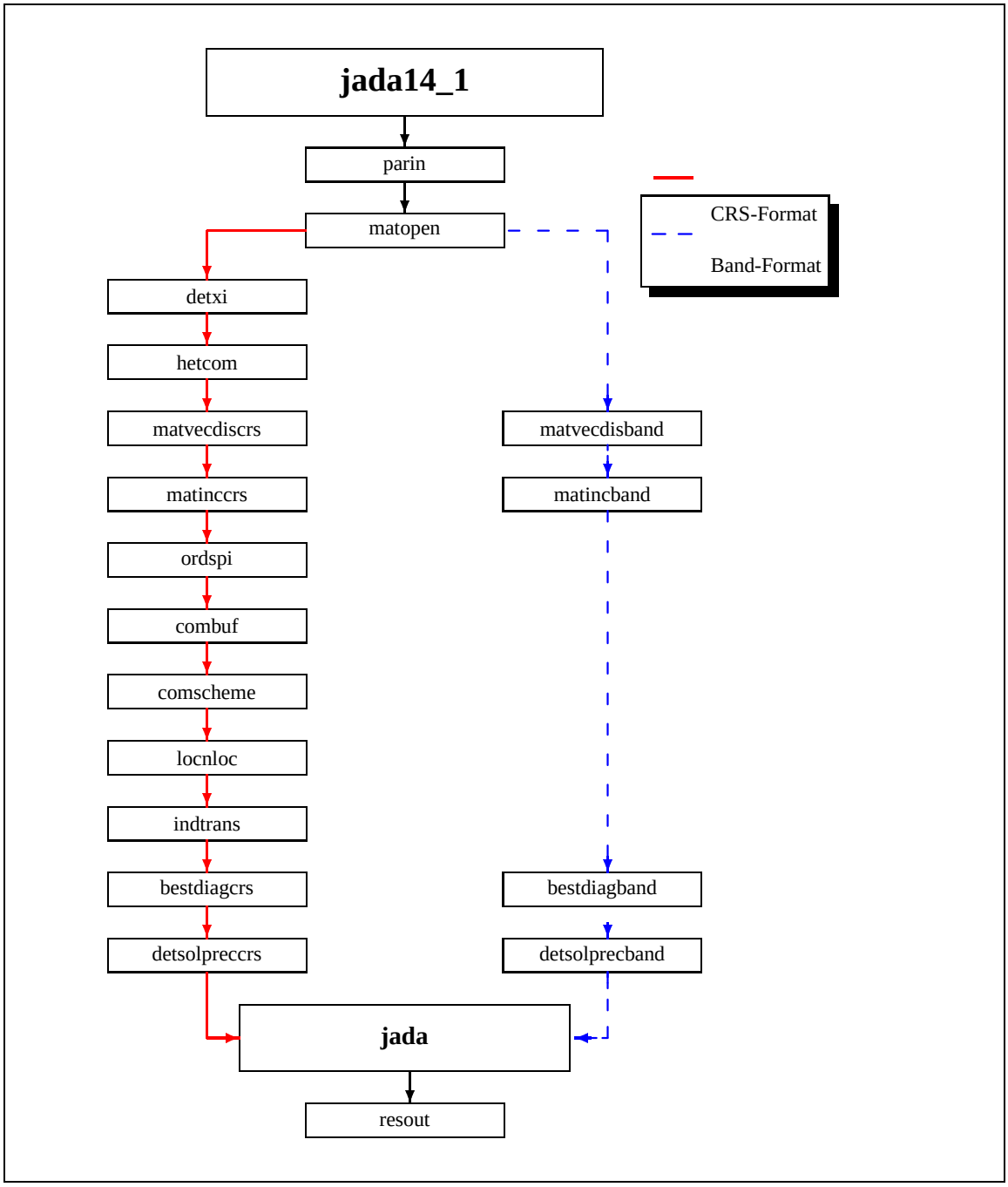


Abbildung 7.1: Modulplan des Programms

Danach wird je nach Matrix-Format ein anderer „Weg“ eingeschlagen und die entsprechenden Funktionen dieses Formats werden aufgerufen. Mit Hilfe von *detxi* wird das ξ (siehe Kapitel 5.3) bestimmt, welches die Verteilung der Zeilen beim CRS-Format genauer beschreibt. Das Unterprogramm *hetcom* liest Gewichte ein, um die Matrix in einer heterogenen Umgebung zu verteilen. Mit Hilfe von *matvecdisxxx* wird analog zu den Kapiteln 5.3 und 5.7 die Verteilung der Matrix und der Vektoren auf die einzelnen Prozessoren berechnet. Die Endung *xxx* wird je nach Format durch „crs“ beziehungsweise „band“ ersetzt. Die Prozedur *matincxxx* liest die Matrix ein und verteilt die Matrix zeilenweise entsprechend der berechneten Verteilung auf die einzelnen Prozessoren.

Im CRS-Format werden nun noch einige weitere Funktionen verwendet um die lokalen und nichtlokalen Blöcke zu erzeugen. Die Subroutine *ordspi* ordnet die Koeffizienten einer Zeile aufsteigend nach dem Spaltenindex. In der Prozedur *combuf* wird anhand der vorhandenen Spaltenindizes in einer Zeile berechnet, wie viele Nachrichten von jedem Prozessor zu jedem anderen gesendet und wie viele von jedem anderen empfangen werden müssen. Durch *comscheme* werden damit die Kommunikationsschemata der Prozessoren generiert.

Mit Hilfe von *locnloc* wird, wie in Kapitel 5.3 beschrieben, die Aufteilung in lokale und nichtlokale Blöcke durchgeführt. Das Unterprogramm *indtrans* modifiziert das Feld der Spaltenindizes so, dass direkt auf den komprimierten Vektor zugegriffen werden kann.

Für beide Formate wird im Unterprogramm *bestdiagxxx* die Diagonale der Matrix auf einen einzelnen Vektor geschrieben. Dies ist sehr wichtig, da diese später für Vorkonditionierer gebraucht wird (siehe Kapitel 7.5). Mit Hilfe von *detsolprecxxx* werden die Vorkonditionierer der CG-Löser ILUT beziehungsweise ILDLT so weit es möglich ist vorberechnet.

Im Folgenden läuft das Programm für beide Formate vollkommen identisch. Im Hauptunterprogramm *jada* wird nun der eigentliche Jacobi-Davidson-Algorithmus durchgeführt. Anschließend werden von der Subroutine *resout* die Endergebnisse auf dem Bildschirm und in Datei ausgegeben.

7.3 Der Jacobi-Davidson-Algorithmus

Das Hauptunterprogramm realisiert den eigentlichen Jacobi-Davidson-Algorithmus. Es gliedert sich in folgende Teilaufgaben auf:

- Berechnung der Startvektoren
- Lösung der Korrekturgleichung
- Erweiterung des Unterraums
- Berechnung der Eigenwerte und -vektoren
- Berechnung der Ritzwerte und -vektoren
- Durchführung eines Restarts

Zur Realisierung der Aufgaben werden einige weitere Unterprogramme verwendet. Der Aufbau wird durch die Abbildung 7.2 verdeutlicht. Zunächst wird Speicher für Iterationsvektoren und -matrizen, so wie Kommunikationspuffer angelegt.

Es werden vier verschiedene Methoden bereit gestellt um Startvektoren $v_i^{(0)}$ zu erzeugen: Mit Hilfe der Subroutine *startvec* werden die ersten Einheitsvektoren als Startvektoren generiert. Weiterhin können die orthogonalen Anteile der Diagonalmatrix zur Generierung der Startvektoren verwendet werden.

Die Prozedur *startdiag* verwendet auch die Informationen der Diagonale, um Startvektoren zu erzeugen. Allerdings werden hier Vielfache der Einheitsvektoren erzeugt, die Längen entsprechend der Diagonaleinträge besitzen und nach diesen sortiert sind. Die vierte Methode erzeugt Pseudo-Zufallsvektoren, die mit Hilfe der Routine *gramsch* nach Gram-Schmidt orthogonalisiert werden. (Angedacht, aber bisher noch nicht implementiert ist hier ein Einlesen vorgegebener Startvektoren.)

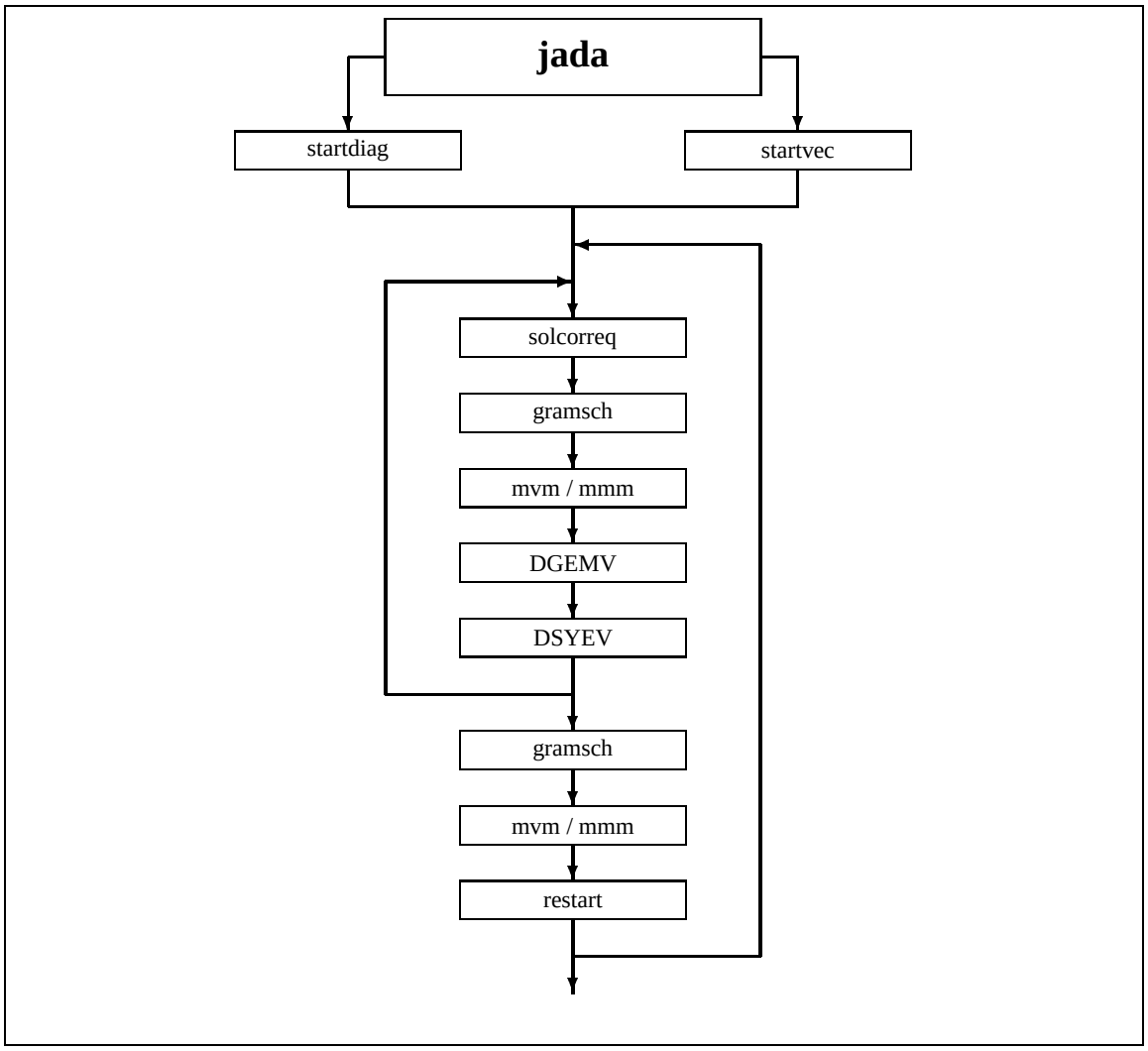


Abbildung 7.2: grober Modulplan des Jacobi-Davidson-Algorithmus

Anschließend werden mit Hilfe des Unterprogramms *mvm* (Matrix-Vektor-Produkt) beziehungsweise *mmm* (Matrix-Matrix-Produkt) die Vektoren $w_j^{(0)} = Av_j^{(0)}$ berechnet. Die Startvektoren $v_j^{(0)}$ und $w_j^{(0)}$ werden normalisiert.

im Folgenden wird nun der Programmablauf für die Suche nach äußeren Eigenwerten symmetrischer Matrizen beschrieben. Dies entspricht dem Ablauf des Programms vor der Diplomarbeit. Die Behandlung von unsymmetrischen Matrizen und inneren Eigenwerten wurde erst im Rahmen der Diplomarbeit implementiert und wird im Kapitel 7.4 erläutert. Als nächstes wird mit Hilfe der BLAS-Routine [15] $DGEMV$ die Matrix $H^{(0)} = (V^{(0)})^T W^{(0)}$ als untere Dreiecksmatrix berechnet. Weiterhin werden Startresiduuen und deren Normen berechnet. Das projizierte Problem wird auf jedem Prozessor gespeichert und seriell verarbeitet.

Nun folgt die eigentliche Iteration: Diese setzt sich aus einer äußeren und einer inneren Iteration zusammen. Zunächst wird durch *solcorreq* die Korrekturgleichung $(I-u_k u_k^T)(A-\theta_k I)(I-u_k u_k^T)v = -r_k$ gelöst. Die Lösung der Gleichung wird durch *gramsch* gegen den Unterraum $V^{(k)}$ orthogonalisiert. Falls die Orthogonalisierung fehl schlägt, wird ein *restart* durchgeführt. Ansonsten werden die Iterationsmatrizen $W^{(k)}$ und $H^{(k)}$ sowie die Residuuen neu berechnet.

Nun werden mit Hilfe der LAPACK-Routine [15] *DSYEV* seriell die Eigenwerte und Vektoren der Matrix $H^{(k)}$ berechnet. Im nächsten Schritt werden die Eigenwerte und -vektoren in die richtige Reihenfolge sortiert. Mit Hilfe der BLAS-Routine *DGEMV* werden die Ritzwerte und Ritzvektoren $u_i^{(k)} = V^{(k)} s_i^{(k)}$ berechnet. Im Folgenden wird nun geprüft, welche Eigenpaare bereits konvergiert

sind. Sind alle Paare konvergiert, so werden beide Iterationen abgebrochen. Weiterhin wird an dieser Stelle geprüft, ob eventuell ein Restart durchgeführt werden muss, weil Vektornormen zu klein geworden sind. Nach der inneren Schleife wird in der äußeren Schleife ein *restart* durchgeführt. Nach den eigentlichen Jacobi-Davidson-Iterationen wird der zu Anfang angelegte Speicher wieder freigegeben.

7.4 Aufbau des Unterraumes

Die Komplexität eines Standard-Eigenwert-Lösers ist $O(n^3)$. Es ist offensichtlich, dass dies für große Matrizen jeden Rahmen sprengen würde. Daher wird das Problem im Jacobi-Davidson-Verfahren in einen Unterraum projiziert. Die Eigenwerte und -vektoren des projizierten Problems stellen dann die Ritzwerte und -vektoren des ursprünglichen Problems dar.

Im Standard Jacobi-Davidson-Verfahren verwendet man die Projektion:

$$H = V^T AV = V^T W$$

Um diese zu erzeugen, wird zunächst eine Basis aus orthonormierten Vektoren für den Raum V generiert. Dann wird die Matrix $W = AV$ berechnet. Damit lässt sich dann $H = V^T W$ berechnen. Da die Spaltenvektoren von V orthonormal sind gilt:

$$\begin{aligned} Hy =& V^T Wy = V^T AVy = \theta y \\ \Leftrightarrow \quad & VV^T AVy = V\theta y \\ \Leftrightarrow \quad & (A - \theta I)(Vy) \perp V \end{aligned}$$

Man erkennt direkt, dass die Eigenwerte von H und die Eigenvektoren multipliziert mit V Näherungen der Eigenwerte und -vektoren von A darstellen.

Das Programm sollte um die Funktion der Suche nach inneren Eigenwerten erweitert werden. Man bezeichnet die Methode als Jacobi-Davidson-Verfahren mit harmonischen Ritzwerten, und verwendet die folgende Projektion:

$$H = ((AV)^T V)^{-1} = (W^T V)^{-1}$$

Bei diesem Verfahren wird eine Basis aus orthonormierten Vektoren für den Raum W erzeugt. Entsprechend der Bedingung $W = AV$ wird dann eine passende Basis des Raums V erzeugt. Dabei lässt sich ein explizites Invertieren von A vermeiden, in dem man die Basen sukzessiv wie folgt aufbaut:

$$w = At_i^k, \quad \tilde{w} = w - W_k^{m_k} {W_k^{m_k}}^T w, \quad \tilde{t} = t_i^k - V_k^{m_k} {W_k^{m_k}}^T w$$

$$w_{m_k+1} = \frac{\tilde{w}}{\|\tilde{w}\|}, \quad v_{m_k+1} = \frac{\tilde{t}}{\|\tilde{w}\|}$$

Dabei sind $W_k^{m_k}$ und $V_k^{m_k}$ die bisher erzeugten Basen der Dimension m_k. Man erhält dann durch Multiplikation der beiden Matrizen $H^{-1} = W^T V$. Diese Matrix muss zur Berechnung der Eigenwerte nicht mehr explizit invertiert werden, denn die Eigenwerte der Inversen einer Matrix sind identisch mit den inversen Eigenwerten der Matrix selbst.

Der Trick an dieser Berechnung ist der folgende: Es sollen Eigenwerte um 0 herum bestimmt werden. Das Jacobi-Davidson-Verfahren konvergiert aber nur schnell für äußere Eigenwerte, also Eigenwerte mit großem Betrag. Da aber nun die Eigenwerte der Matrix H^{-1} und nicht der Matrix H bestimmt werden und man somit anstatt den eigentlichen Eigenwerte ihre Inversen erhält, sind somit die Eigenwerte um 0 herum die betragsmäßig größten. Somit hat man wieder den gleichen Fall wie beim Standard-Verfahren.

Möchte man nun innere Eigenwerte um einen Wert τ herum berechnen, so führt man einen Shift ein und berechnet W als $W = (A - \tau I)V$. Damit erhält man dieselbe Situation wie beim Wert „0“. Allerdings muss man die Eigenwert-Näherungen dann wieder entsprechend zurück transformieren, wie die folgende Rechnung zeigt. Weiterhin ist zu beachten, dass gelten muss $A(Vy) \approx \theta(Vy)$.

$$\begin{aligned} Hy = (W^T V)^{-1} y \approx & V^+ (W^T)^+ y \approx V^+ W y = V^+ (A - \tau I) V y = \theta y \\ \Leftrightarrow \quad & V V^+ (A - \tau I) V y = V \theta y \\ \Leftrightarrow \quad & (A - (\theta + \tau) I)(Vy) \perp V \end{aligned}$$

Neben der Unterscheidung zwischen inneren und äußeren Eigenwerten ist beim Aufbau des Unterraumes zu beachten, ob die Systemmatrix A symmetrisch ist. Ist dies der Fall, so ist auch die projizierte Matrix H symmetrisch, denn es gilt:

$$H^T = (V^T W)^T = W^T V = (AV)^T V = V^T A^T V = V^T A V = V^T W = H$$

beziehungsweise für den harmonischen Fall:

$$\begin{aligned} H^T &= (W^T V)^{(-1)^T} = (W^T V)^{T^{-1}} = (V^T W)^{-1} = (V^T A V)^{-1} \\ &= (V^T A^T V)^{-1} = ((AV)^T V)^{-1} = (W^T V)^{-1} = H \end{aligned}$$

Ist H symmetrisch, so genügt es das untere Dreieck von H abzuspeichern. Weiterhin kann ein Eigenwertlöser für symmetrische Matrizen verwendet werden, der sehr viel schneller arbeitet als ein Löser für unsymmetrische Matrizen. Im unsymmetrischen Fall muss die Matrix H als volle Matrix abgespeichert und ein unsymmetrischer Eigenwert-Löser verwendet werden.

7.5 Lösung der Korrekturgleichung

Das Jacobi-Davidson-Verfahren besitzt verschiedene Freiheitsgrade. Den einen stellt das Verfahren zur Berechnung der Eigenwerte der Iterationsmatrix $H^{(k)}$ dar. Diese werden sehr effizient mittels den LAPACK-Routinen *DSYEV* / *DGEEV* berechnet. Hier könnte man gegebenenfalls auch ein anderes Verfahren einsetzen. Viel entscheidender ist aber die Wahl des Lösers der Korrekturgleichung.

Hier gibt es nun sehr viele verschiedene Möglichkeiten, die sehr starken Einfluss auf die Konvergenzgeschwindigkeit haben. Es gibt zwei Extremfälle: Zum einen kann man einen sehr schlechten aber sehr schnellen Löser verwenden, was natürlich dazu führt, dass eine Iteration sehr schnell läuft, die Konvergenz allerdings sehr langsam ist. Das genaue Gegenteil stellen sehr präzise aber damit auch aufwändige Löser dar. Hier hat man den Effekt, dass eine Iteration die vielfache Zeit eines primitiven Lösers in Anspruch nimmt. Der Vorteil ist natürlich, dass wesentlich weniger Iterationen benötigt werden. Einen großen Vorteil erhält man nun durch die Kombination dieser beiden Fälle. Das heißt Man beginnt mit einem schnellen, schlechten Löser und führt mit diesem eine feste Anzahl Iterationen durch, danach wechselt man zu einem präzisen Löser. Es liegt auf der Hand, dass dies von Vorteil ist, denn wenn die Näherung noch sehr weit von der Lösung weg liegt, lohnt es sich nicht, einen sehr präzisen Löser zu verwenden. Umgekehrt benötigt ein einfacher Löser bei Näherungen sehr nah an der exakten Lösung viel zu viele Iterationen um eine Näherung zu berechnen, die genau genug ist. Hier lohnt sich also der Einsatz von präzisen Lösern.

Die beschriebenen Möglichkeiten stehen auch im Programm zur Verfügung. Als einfacher Löser wird die Multiplikation mit der Inversen der Diagonal- beziehungsweise einer inneren Bandmatrix verwendet. Diese Aufgabe wird vom Unterprogramm *simplesolv* übernommen. Die Prozedur *complsolv* stellt als komplexere Löser die CG-artigen Verfahren QMR und TFQMR zur Verfügung. Weiterhin besteht noch die Möglichkeit diese mit den unvollständigen Gaußzerlgengungen ILDLT und ILUT vorzukonditionieren.

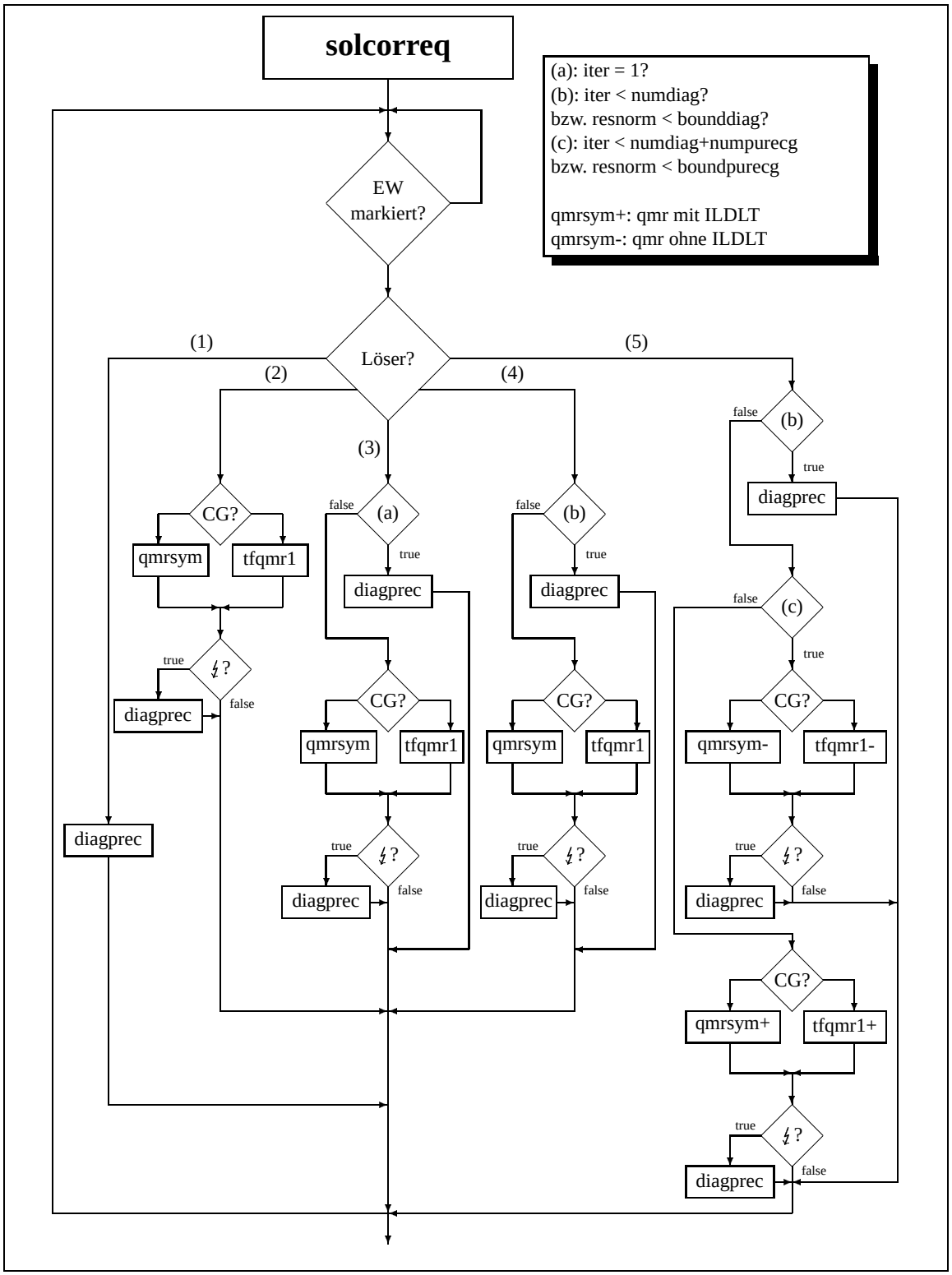


Abbildung 7.3: grober Ablaufplan zur Lösung der Korrekturgleichung

Insgesamt stehen dem Nutzer nun folgende Möglichkeiten zur Verfügung:

1. Diagonal- beziehungsweise Bandlöser (DL)
2. QMR / TFQMR (CG) mit / ohne ILDLT / ILUT (VK)
3. DL als Startiteration, dann CG mit / ohne VK
4. zunächst DL, dann CG mit / ohne VK
5. zunächst DL, dann CG ohne VK, dann CG mit VK

Der Wechsel des Lösers anhand der Iterationszahl kann nur dann optimal genutzt werden, wenn die Wechselpositionen sehr gut gewählt sind. Dazu muss der Benutzer aber die Struktur der Matrix sehr genau kennen. Im Rahmen der Diplomarbeit wurde daher eine weitere Option implementiert, die dafür sorgt, dass der Wechsel anhand der Residuuen-Norm (im Verhältnis zur Anfangsnorm) durchgeführt wird. Damit hat man eine Angabe zur Verfügung, die für jede Matrixstrukur die gleiche Aussagekraft besitzt und der Benutzer muss somit die Matrix-Struktur nicht genauer kennen.

Die Lösung der Korrekturgleichung wird von der Funktion *solcorreq* übernommen. In dieser wird anhand eines eingelesenen Parameters entschieden, welcher Löser verwendet werden soll. Weiterhin wird über Parameter gesteuert, ob und wenn ja welcher Vorkonditionierer verwendet werden soll, wie viele „fill ins“ für diesen verwendet werden sollen und nach wie vielen Schritten gegebenenfalls zwischen den Lösern gewechselt werden soll. Eine genaue Beschreibung der Parameter ist in Kapitel 8.1 gegeben.

Der Aufbau der Funktion *solcorreq* wird durch die Abbildung 7.3 verdeutlicht. Bei den verwendeten CG-artigen Verfahren kann es zu Abbrüchen, so genannten „Break-Downs“ kommen. Tritt ein Break-Down auf, so wird der entsprechende einfache Löser verwendet.

Beim Standard-Ritz-Verfahren muss die Korrekturgleichung

$$(I - u_k u_k^T)(A - \tilde{\theta}_k I)(I - u_k u_k^T) = -r_k$$

gelöst werden. Beim harmonischen Ritz-Verfahren ändert sich dadurch, dass hier nicht mehr alle Orthogonalitätsbeziehungen gelten die rechte Seite

$$(I - u_k u_k^T)(A - \tilde{\theta}_k I)(I - u_k u_k^T) = -(I - u_k u_k^T)r_k$$

Da die Korrekturgleichungen im Wesentlichen übereinstimmen, können natürlich auch dieselben Löser verwendet werden. Den Lösern wird lediglich die modifizierte rechte Seite übergeben.

Zur Berechnung der rechten Seite muss weder die Matrix explizit aufgestellt noch eine Matrix-Vektor-Multiplikation durchgeführt werden, wie die folgende Rechnung zeigt.

$$(I - u_k u_k^T)r_k = r_k - u_k u_k^T r_k = r_k - < u_k, r_k > u_k$$

Verwendet man bei der Berechnung des Residuum anstatt der aktuellen Eigenwertnäherung den entsprechenden Rayleigh-Quotienten, so gelten die Orthogonalitätsbeziehungen weiterhin. Dadurch ist es möglich weiterhin mit der Korrekturgleichung des Standard-Ritz-Verfahren arbeiten.

Zur Beschleunigung der Konvergenz ersetzt man bei den ersten Iterationen die Näherung θ durch das Ziel τ. Natürlich muss dann auch das Residuum der rechten Seite mit Hilfe von τ berechnet werden. Die neue rechte Seite r_k^{tau} steht nun nicht mehr senkrecht auf u_k. Da dies aber vorausgesetzt wird, muss auch diese in den Unterraum senkecht zu u_k projiziert werden. Die rechte Seite, die sich daraus ergibt, entspricht dem über den Rayleighquotienten berechneten Residuum:

$$\begin{aligned} r_k^{tau} &= (I - u_k u_k^T)(A - \tau I)u_k \\ &= (I - u_k u_k^T)(Au_k - \tau u_k) \\ &= Au_k - u_k u_k^T Au_k - \tau u_k + u_k u_k^T \tau u_k \\ &= Au_k - u_k < u_k, Au_k > -\tau u_k + \tau u_k < u_k, u_k > \\ &= Au_k - u_k < u_k, Au_k > \end{aligned}$$

7.6 Bandlöser

Für Bandmatrizen stellt die Inverse einer Matrix, die nur ein paar innere Bänder der Originalmatrix enthält, einen Näherungslöser dar. Dieser wurde im Rahmen der Arbeit implementiert.

Die Inverse kann mit Hilfe einer LU-Zerlegung der Matrix berechnet werden. Diese lässt sich aber auf einem Parallelrechner nicht ohne weiteres effizient durchführen, da dazu einzelne Dreiecksblöcke der Matrix unter den Prozessoren ausgetauscht werden müssen. Daher wurde zur Berechnung der Inversen die Library „ScaLAPACK“ [16] verwendet. Diese stellt parallele Funktionen bereit, die Aufgaben analog zu den „LAPACK“-Funktionen [15] übernehmen.

Problematisch dabei ist zunächst, dass „SCALAPACK“ eine spaltenweise Matrixspeicherung verwendet, während im Programm eine zeilenweise Speicherung (siehe Kapitel 5.5) verwendet wird.

Beispiel 17 *Sei*

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 6 & 7 & 8 & 9 \\ 0 & 10 & 11 & 12 \end{pmatrix}$$

dann ergibt sich als Speicherung:

LAPACK:	*0*	*1*	*3*	*6*	*2*	*4*	*7*	*10*	*0*	*5*	*8*	*11*	*9*	*12*	*0*	*0*
Programm:	*0*	*0*	*1*	*2*	*0*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*0*

Diese Probleme behebt die Funktion *detsolprec*. Dort wird aus der wie in Kapitel 5.5 beschriebenen Format gespeicherten Matrix eine schmalere Bandmatrix mit n_l Sub- und n_r Superdiagonalen erzeugt, die im „ScaLAPACK“-Format abgespeichert wird. Im symmetrischen Fall reicht es aus, die inneren Bänder der Matrix zu übernehmen und in eine Matrix im „ScaLAPACK“-Format zu übernehmen.

Beispiel 18 *Sei*

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 2 & 3 & 4 & 0 \\ 0 & 4 & 5 & 6 \\ 0 & 0 & 6 & 7 \end{pmatrix}$$

dann ergibt sich als Speicherung:

LAPACK:	*0*	*1*	*2*	*2*	*3*	*4*	*4*	*5*	*6*	*6*	*7*	*0*
Programm:	*0*	*1*	*2*	*2*	*3*	*4*	*4*	*5*	*6*	*6*	*7*	*0*

Im Programm wird die Matrix zeilenweise aufgeteilt, im „ScaLAPACK“ Format spaltenweise. Es lässt sich leicht erkennen, dass im symmetrischen Fall, unabhängig davon ob die Matrix zeilen- oder spaltenweise aufgeteilt wird, immer die gleichen Koeffizienten auf jedem Prozessor liegen.

Im unsymmetrischen Fall wird es komplizierter, denn hier wird die transponierte Matrix benötigt, um die Matrix im „ScaLAPACK“-Format zu speichern, und daher müssen Komponenten der Matrix unter den einzelnen Prozessoren ausgetauscht werden. Wie in Kapitel 5.8 gezeigt wurde, reduziert sich die Kommunikation der Prozessoren aber auf die jeweiligen Nachbarprozessoren.

Der eigentliche Löser, in dem die Inverse der Matrix berechnet wird, ist im Unterprogramm *bandprec* implementiert. Die Berechnung der Inversen, mit Hilfe einer LU-Zerlegung, übernimmt die „ScaLAPACK“ Routine *PDGBSV* (Parallel Double General Band Solver). Diese benötigt unter anderem zwei Argumente, die die Aufteilung der Matrix und der rechten Seite beschreiben. Der Ganzzahlvektor *desca* beschreibt die Aufteilung der Systemmatrix. Die Verteilung der rechten Seite

wird durch *descb* beschrieben. Die beide Vektoren bestehen aus sieben Ganzzahlwerten mit folgenden Bedeutungen:

Index	Beschreibung	*desca*	*descb*
1	Verteilung	501 = spaltenw.	502 = zeilenw.
2	BLACS-Kontext		
3	globale Anzahl an	Spalten	Zeilen
4	lokale Anzahl an	Spalten	Zeilen
5	Prozessorindex mit erster	Spalte	Zeile
6	Länge einer lokalen	Spalte	Zeile
7	unbenutzt, aber reserviert		

7.7 Unterscheidung der Formate

Im Kapitel 5 wurde bereits erwähnt, dass das Programm so umstrukturiert wurde, dass außer bei Matrix-Vektor-Operationen nur an einer Stelle eine Unterscheidung zwischen den einzelnen Formaten stattfindet. Insgesamt gibt es eigentlich drei Stellen in denen nach dem Matrix-Format unterschieden wird. Bei der Verteilung der Matrizen und Vorbereitung der Vorkonditionierer ist natürlich das Format der Matrizen entscheidend. Deshalb muss hier für jedes Format separater Code zur Verfügung gestellt werden. An dieser Stelle werden aber auch im Programm je nach Format unterschiedliche Datenstrukturen allokiert, daher ist es notwendig am Ende des Hauptprogramms noch einmal Format-spezifisch Speicher freizugeben. Diese Aufgaben übernimmt das Hauptprogramm. Das Jacobi-Davidson-Verfahren an sich verwendet nur Matrix-Vektor-Produkte und benötigt somit keinen weiteren Zugriff auf die Matrix. Anders sieht es aus, wenn man die Lösung der Korrekturgleichung betrachtet. Hier kann es nötig sein, dass die Matrix anderweitig benötigt wird. Betrachtet man die Löser genauer, so stellt man fest, dass bei der Multiplikation mit der Inversen der Diagonalmatrix lediglich die Diagonale der Matrix benötigt wird. Diese kann man aber von vorn herein auf einem Feld speichern, so dass nachher nur noch dieses benötigt wird und das ursprüngliche Matrix-Format keine Rolle mehr spielt. Allerdings eignet sich für Bandmatrizen ein Bandlöser besser. Daher bietet es sich an, je nach Format einen anderen einfachen Löser zur Verfügung zu stellen. Als komplexere Löser eigenen sich für jedes Format die CG-artigen Verfahren. Bei diesen lässt sich feststellen, dass diese auch nur in Form von Matrix-Vektor-Produkten auf die Matrix zugreifen. Routinen dafür stehen aber bereits, über die in Kapitel 6.1 beschriebene Schnittstelle, zur Verfügung. Die Vorkonditionierer lassen sich im vorhinein so vorbereiten, dass man später im Programm keine Format-spezifische Unterscheidung mehr benötigt. (Diese wäre dann erforderlich, wenn man für jedes Format eine speziell für dieses Format optimierte Implementierung verwenden möchte. Im Programm wird bisher als Format für die Zerlegung ein Format für dünn-besetzte Matrizen verwendet, die natürlich eine Obermenge der Bandmatrizen darstellen)

Insgesamt ergeben sich also folgende drei Schnittstellen zur Unterscheidung des Formats:

1. Aufteilung der Systemmatrix -> Hauptprogramm
2. Berechnung eines Matrix-Vektor-Produktes -> Matrix-Modul
3. Lösung der Korrekturgleichung -> Löser-Modul

Es lässt sich leicht erkennen, dass der eigentliche Jacobi-Davidson-Algorithmus somit vollkommen unabhängig vom verwendeten Format wird.

7.8 Einbinden neuer Formate

Durch die im vorherigen Kapitel beschriebene Organisation lassen sich auch sehr leicht neue Matrix-Formate einbinden. Zunächst benötigt man entsprechende Routinen für das jeweilige Format, die

die Matrix auf die einzelnen Prozessoren aufteilt. Weiterhin benötigt man ein Unterprogramm, das ein Matrix-Vektor-Produkt im jeweiligen Format berechnen kann. Um bessere Performance zu erreichen sollte zusätzlich eine Routine für ein Matrix-Matrix-Produkt implementiert werden.

Möchte man den einfachen Löser, der mit der Inversen der Diagonale multipliziert, beziehungsweise die Vorkonditionier der CG-artigen Verfahren verwenden, so müssen zusätzlich Unterprogramme erstellt werden, die die Diagonale auf einen Vektor schreiben beziehungsweise die Zerlegungen vorbereiten.

Im Hauptprogramm muss nun ein zusätzlicher „case“ bei der Auswahl des Matrix-Formates eingebaut werden. In diesem werden dann die entsprechenden Funktionen aufgerufen. Weiterhin muss hier der Speicher für die benötigten Datenstrukturen allokiert werden.

Verwendet man zur Beschreibung der Matrix neue, bisher unbekannte, Variablen, so müssen diese zusätzlich im Matrix-Modul *matrix_mod* eingetragen werden. Zwingend erforderlich ist, dass man auch in den Funktionen *mvm* beziehungsweise *mmm* einen neuen „case“ einführt, in dem die entsprechenden Routinen, die das Matrix-Vektor- beziehungsweise Matrix-Matrix-Produkt für das neue Format bereitstellen, aufgerufen werden.

Das Format sollte natürlich auch in der Parameterdatei *jadaxxx.in* eingetragen werden.

Zusammenfassend ergeben sich also folgende Aufgaben beim Einbinden eines neuen Formats:

- Implementierung der Matrix-Aufteilung
- Implementierung von Vorbereitungsroutinen für Löser und Vorkonditionierer
- Einfügen der Routinen in einen neuen „case“ im Hauptprogramm
- Einfügen neuer Variablen in das Matrix-Modul *matrix_mod*
- Implementierung von Matrix-Vektor- und Matrix-Matrix-Produkt
- Einfügen der Produkte in die Schnittstellenroutinen *mmm* und *mvm*
- Einfügen des Formats in die Parameterdatei

7.9 Umstellung auf Fortran 90

Das Programm sollte von Fortran 77 auf Fortran 90 umgestellt werden. Dabei sollte möglichst wenig am bestehenden Code verändert werden. Trotzdem haben sich einige wesentliche Veränderungen ergeben, die in diesem Kapitel nun kurz zusammengefasst werden.

Fortran 77 stellte keine dynamische Speicherverwaltung zur Verfügung. Daher wurde der Speicher in der bestehenden Version mit C-Routinen allokiert. Dies hatte allerdings zur Folge, dass im gesamten Programm bei Feldzugriffen jeweils ein Offset mit angegeben werden musste, so dass der Anfang des Feldes im Speicher bekannt war. Dies führte teilweise zu sehr unübersichtlichem Code. Da ab Fortran 90 die dynamische Speicherverwaltung unterstützt wird, wurde im gesamten Programm die Speicherverwaltung umgestellt. Das Programm ist dadurch deutlich übersichtlicher und auch effizienter geworden, da nicht mehr länger C-Routinen aufgerufen werden müssen.

Einen weiteren Vorteil erhält man dadurch, dass man externe Unterprogramme in Module einbindet und sie zu so genannten „Modul-Unterprogrammen“ macht. Dies hat den großen Vorteil, dass die Funktionsschnittstellen damit explizit werden und der Compiler damit überprüfen kann, ob die Parameterübergabe beim Aufruf von Unterprogrammen korrekt ist. Um auch diesen Vorteil von Fortran 90 nutzen zu können, wurden alle Unterprogramme in Module eingebunden.

7.10 Jump-spezifischer Code und Probleme

Das Programm wurde so entwickelt, dass es vollständig portabel ist. Lediglich an einer Stelle im Quellcode wurden aufgrund der Benutzerfreundlichkeit zwei Funktionen aus einer IBM „Compiler-

Extension“ verwendet. Die Funktionen werden im Unterprogramm *matopen* verwendet. Hier wird dem Benutzer die Möglichkeit zur Verfügung gestellt, den Namen der Matrix-Eingabedatei als Argument beim Programmaufruf zu übergeben. Fortran stellt dazu zunächst keine Routinen zur Verfügung. In der Extension „XLFUTILITY“ werden unter anderen die folgenden Funktionen zur Verfügung gestellt.

```
getarg(i1, c1)
    i1: INTEGER(4),INTENT(IN)
    c1: CHARACTER(*),INTENT(OUT)

iargc() result(i1)
    i1: INTEGER(4)
```

Abbildung 7.4: Befehle der XLF-Compiler-Extenion

Der Funktion *getarg* übergibt man den Index *i1* des Arguments, das eingelesen werden soll und sie liest das Argument auf den übergebenen String *c1* ein. Dabei entspricht Argument ’0’ dem Programmnamen. Die Funktion *getarg* gibt die Anzahl der übergebenen Argumente zurück.

Soll das Programm auf einen Rechner portiert werden, der diese Extension nicht unterstützt, kann die entsprechende Stelle im Code einfach auskommentiert werden. Dem Benutzer stehen dann immer noch die beiden anderen Möglichkeiten (siehe Kapitel 8.2) zur Wahl der Matrix zur Verfügung. Eine andere Möglichkeit ist die Verwendung der in Fortran 2003 aufgenommenen Funktion *GET_COMMAND_ARGUMENT*. Da diese aber bisher nur von sehr wenigen Compilern unterstützt wird, wurden die Extension-Funktionen verwendet.

Im Zusammenhang mit dem Einlesen des Dateinamens ist auch ein anderes kleineres Problem aufgetreten. Dem Benutzer wird die Möglichkeit zur Verfügung gestellt den Dateinamen während der Ausführung einzugeben. Dazu sollte er über den Text „filename: “ aufgefordert werden. Üblicherweise erfolgt nach dieser Ausgabe kein Zeilenvorschub, so dass der Name direkt dahinter eingegeben werden kann. Es hat sich allerdings gezeigt, dass die dazu nötige Option „advance = ’no’“ vom mpxlf-Compiler anders umgesetzt wird. Die Option führt bei dem Compiler dazu, dass die Ausgabe erst dann geschrieben wird, wenn eine weitere Ausgabe mit Zeilenvorschub gemacht wird. Dies führt dazu, dass der Rechner auf die Eingabe des Benutzers wartet, dieser allerdings noch keine Aufforderung dazu sieht. Wahrscheinlich blockiert mpxlf die direkte Ausgabe, da die Ausgabe-Reihenfolge der einzelnen Prozessoren nicht gesichert ist. Aus diesem Grund wird die Aufforderung mit Zeilenvorschub ausgegeben, so dass der Benutzer seine Eingabe in der nächsten Zeilen machen muss.

7.11 Übersicht über die Funktionen

In der folgenden Tabelle wird eine kurze alphabetisch geordnete Übersicht über die Aufgaben der im Programm verwendeten Unterprogramme und Module gegeben. Während bisher nur die Haupt-Routinen erläutert wurden, werden hier jetzt alle wichtigen im Programm verwendeten Routinen kurz erklärt. Zahlreiche Sortierroutinen oder andere Hilfsprogramme wurden auf Grund der Übersichtlichkeit bewusst weggelassen. Die Aufgabe dieser geht jeweils aus der Inline-Dokumentation hervor.

Unterprogramm	Aufgabe
bandprec	löst das Vorkonditionierersystem M mit dem Vorkonditionierer $B_A - \theta_k I$ (B_A = innere Bänder von A)
bestdiagxxx	schreibt die Diagonale einer Matrix im Format xxx in einen Vektor
bfile_read	liest Ganz-, doppelt genaue Gleitkommazahlen, Zeichen und Vektoren dieser Datentypen aus einer Binärdatei ein
combuf	bestimmt wie viele Nachrichten gesendet und empfangen werden müssen
complsolv	stellt einen aufwändigen Löser für die Korrekturgleichung zur Verfügung
comscheme	erstellt ein Kommunikationsschema
detsolprec	berechnet Löser und Vorkonditionierer vor
detxi	berechnet das ξ für die Verteilung der Matrix
DGEEV (LAPACK)	berechnet die Eigenwerte und Vektoren einer unsymmetrischen Matrix
DGEMM (BLAS)	berechnet ein Matrix-Matrix-Produkt für lokale Matrizen
DGEMV (BLAS)	berechnet ein Matrix-Vektor-Produkt für lokale Matrizen
diagprec	löst das Vorkonditionierersystem M mit dem Vorkonditionierer $D_A - \theta_k I$ (D_A = Diagonale von A)
DSYEV (LAPACK)	berechnet die Eigenwerte und Vektoren einer symmetrischen, als untere Dreiecksmatrix gespeicherten, Matrix
gramsch	führt eine modifizierte Gram-Schmidt-Orthogonalisierung durch
gramsch2	führt den Aufbau der Unterräume bei harmonischen Ritzwerten durch
hetcom	liest Gewichte zur Verteilung der Matrix auf einem heterogenen System aus einer Datei ein
indtrans	transformiert die Spaltenindizes, so dass direkt auf den komprimierten Vektor zugegriffen werden kann
isget	berechnet eine unvollständige symmetrische Gauß-Zerlegung
ilut	berechnet eine unvollständige unsymmetrische Gauß-Zerlegung
jada	realisiert den Jacobi-Davidson-Algorithmus
lmmmcrs	berechnet den lokalen Teil eines Matrix-Matrix-Produkts für eine Matrix im CRS-Format
lmvmcrs	berechnet den lokalen Teil eines Matrix-Vektor-Produkts für eine Matrix im CRS-Format
locnloc	ordnet eine Matrix in lokale und nicht-lokale Blöcke um
lusol	löst ein Gleichungssystem mit der durch *ilut* erzeugten Zerlegung
lusol_isget	löst ein Gleichungssystem mit der durch *isget* erzeugten Zerlegung
matincxxx	liest eine Matrix im Format xxx aus einer Datei ein und sendet die Zeilen an die entsprechenden Prozessoren
matopen	öffnet die durch die Parameter spezifizierte Binärdatei aus der die Matrix eingelesen werden soll
matvecdisxxx	berechnet die Aufteilung einer Matrix im Format xxx
mmm	berechnet ein Matrix-Matrix-Produkt für eine Matrix mit beliebigem Format

mmmband	berechnet ein Matrix-Matrix-Produkt für eine Matrix im Band-Format
mvm	berechnet ein Matrix-Vektor-Produkt für eine Matrix mit beliebigem Format
mvmband	berechnet ein Matrix-Vektor-Produkt für eine Matrix im Band-Format
nlmmmcrs	berechnet den nicht-lokalen Teil eines Matrix-Matrix-Produkts für eine Matrix im CRS-Format
nlmvm	berechnet den nicht-lokalen Teil eines Matrix-Vektor-Produkts für eine Matrix in beliebigem Format
nlmvmcrs	berechnet den nicht-lokalen Teil eines Matrix-Vektor-Produkts für eine Matrix im CRS-Format
ordspi	ordnet die Indizes der Spalten in jeder Zeile in aufsteigender Reihenfolge an und verschiebt Elemente mit lokalem Zugriff an den Anfang der Zeilen
parin	liest die Parameter des Programms aus einer Datei ein und versendet sie an alle Prozessoren
PDGBSV (ScaLAPACK)	löst ein Gleichungsystem $Ax = b$, wobei A eine Bandmatrix darstellt
qmrsym	löst iterativ $(I - uu^T)(A - \theta_k I)t = -r$ mit Hilfe der Quasi-Minimal-Residual-Methode
resout	gibt die Ergebnisse des Programms auf dem Bildschirm aus und schreibt sie in Datei
restart	realisiert den Restart des Jacobi-Davidson-Verfahrens
simplesolv	stellt einen einfachen Löser für die Korrekturgleichung zur Verfügung
startvec	erzeugt orthogonale Einheitsvektoren als Startvektoren
startdiag	berechnet Startvektoren anhand der Diagonalen der Matrix
tfqmr1	löst iterativ $(I - uu^T)(A - \theta_k I)t = -r$ mit Hilfe der Transpose-Free Quasi-Minimal-Residual-Methode

Tabelle 7.1: Übersicht der wichtigen Unterprogramme

Kapitel 8

Benutzeranleitung

8.1 Parameterdatei

Das Programm wird über die Parameterdatei *jadaxxx.in* gesteuert. Vor jedem Aufruf - besonders vor dem ersten - sollte überprüft werden, ob hier alle Optionen korrekt gesetzt sind und gegebenenfalls Optionen neu gesetzt werden. Folgende Optionen können in der Datei festgelegt werden:

Option	Wertemenge
Art der zu suchenden Eigenwerte	0 = größte, 1 = kleinste 2 = innere
# zu suchenden Eigenwerte	$\leq$ max. # Iter. vor Restart
# Bereich für innere EW	$\in \mathbb{R}$
max. # Iterationen vor Restart	≥ 1
# Iterationen des Lösers	≥ 1
Inkrementierung der # Löser-Iterationen	0 = nein 1 = $\hat{=}$ Iterationsschritt, 2 = $\hat{=}$ letztes Residuum
Abbruchgrenze: Residuennorm / Anfangsnorm	> 0.0
Matrixformat	1 = CRS, 2 = Band
Matrix symmetrisch?	0 = nein, 1 = ja
MM-Produkt verfügbar?	0 = nein, 1 = ja
Methode zur Startvektor-Generierung	0 - 3 (siehe 7.2)
iterativer Löser	0 = TFQMR, 1 = QMR
Löser	0 - 3 (siehe 7.2)
# Bänder	≥ 1
Methode zum Löserwechsel	0 = $\hat{=}$ Iterationsschritt, 1 = $\hat{=}$ letztes Residuum
# Iterationen des Diagonalvorkonditionierers	≥ 0
# Iterationen des CG-Verfahrens ohne Vorkonditionierer	≥ 0
Residuumgrenze für Diagonalvorkonditionierer	≥ 0.0
Residuumgrenze für CG-Verfahren ohne Vorkonditionierer	≥ 0.0

Vorkonditionierer für den Löser	0 = keiner, 1 = ILUT, 2 = ILDLT
# Fill-ins	≥ 0
Fill-in Strategie	0, 1 (siehe Parameterdatei)
Verwerfungsgrenze	> 0.0
Startvektor des Lösers	0 = Nullvektor, 1 = letzte Näherung
ξ für Diagonalvorkonditionierer	≥ 0.0
ξ für iterativen Löser	≥ 0.0
Berechnung von ξ zur Laufzeit	0 = nein, 1 = ja
Verwendung von Prozessor-Gewichtungen	0 = nein, 1 = ja, lese Gewichte aus *weights.in*
Verwendung von Cuthill McKee-Bandreduktion	0 = nein, 1 = ja
Bildschirmausgabelevel	0 = keine, ..., 4 = alles
Index der einzulesenden Matrix	> 0 bzw. 0 = lese Dateinamen ein
max. # Jacobi-Davidson-Iterationen	≥ 1
max. # Iterationen des iterativen Lösers	≥ 1

Tabelle 8.1: Parameter des Programmes

Bei der Änderung der Datei ist darauf zu achten, dass jeweils in jeder Zeile, die keinen Parameter enthält ein „*“ in der ersten Spalte stehen muss. Ist dies nicht der Fall, so werden vom Programm Nullen als Parameter gelesen.

8.2 Matrix-Dateien

Das Programm erwartet als Eingabedateien jeweils Binärdateien der Satzlänge 8 Byte. Für das CRS-Format sollte diese folgenden Aufbau besitzen:

Satz	Inhalt
1 - 3	Titel der Matrix
4	Dimension der Matrix
5	maximale Anzahl an Nicht-Null-Elementen pro Zeile
6 ...	Zeilenbeginne
...	Koeffizienten
...	Spaltenindizes

Tabelle 8.2: Aufbau einer CRS-Eingabedatei

Beim Bandmatrizen-Format wird die folgende Syntax erwartet:

Satz	Inhalt
1 - 3	Titel der Matrix
4	Dimension der Matrix
5	Anzahl der Subdiagonalen
6	Anzahl der Superdiagonalen
7 ...	Koeffizienten

Tabelle 8.3: Aufbau einer Band-Matrix-Eingabedatei

Dem Benutzer stehen verschiedene Möglichkeiten zur Wahl der Matrix-Datei zur Verfügung:

1. Angabe des Index der Matrix in der Parameterdatei
2. Übergabe des Dateinamens als Parameter beim Programmaufruf
3. Eingabe des Dateinamens während der Programmausführung

Für Matrizen, die mehrfach verwendet werden, empfiehlt sich die erste Variante, bei der die Matrix-Datei im Unterprogramm *matopen* anhand eines Index zum Lesen geöffnet wird. Das heißt wenn man eine neue Eingabedatei verwenden möchte, so muss man diese mit in das Unterprogramm *matopen* aufnehmen. Weiterhin sollte man eine kleine Beschreibung der Matrix in die Parameterdatei einfügen. In dieser muss man nun den entsprechenden Index der neuen Matrix eintragen, um mit dieser zu arbeiten.

Eine neue Matrix-Datei wird wie folgt eingefügt: In der Datei *matopen.f* muss im „Select-Befehl“ *select(nrgls)* ein neuer „Case“ eingefügt werden. im Folgenden Beispiel wird die Datei „bcsstk01_crs.out“ mit dem Index 200 eingefügt:
Beispiel: **case** 200: filename = „./MatrixCreator/bcsstk01_crs.out“

Weiterhin sollte in der Parameterdatei eine kurze Beschreibung der Matrix eingefügt werden:
Beispiel: * 200: HARBO, BCSSTK01, n=48, non-zeros=224, CRS

Dies klingt zunächst etwas umständlich, hat aber den Vorteil, dass man nicht bei jedem Aufruf den Dateinamen angeben muss, sondern lediglich in der Parameterdatei die korrekte Nummer anzugeben braucht. Weiterhin hat man in der Parameterdatei eine Art Archiv der Matrizen und ein lästiges Suchen nach einer bestimmten Matrix entfällt.

Möchte man eine Matrix nur ein Mal verwenden, so bieten sich eher die beiden anderen Varianten an. Dazu setzt man in der Parameterdatei den Matrix-Index auf „0“. Nun kann man entweder den Dateinamen der Matrix-Datei als Argument beim Aufruf mit übergeben oder man wird während der Programmausführung dazu aufgefordert einen Dateinamen einzugeben. Es wird erwartet, dass die Dateien im Verzeichnis „MatrixCreator“ liegen. Dieses braucht (darf) allerdings nicht mit angegeben zu werden.
Beispiel: Beim Aufruf „llrun -p16 jada14_1.exe bcsstk13_crs.out“ wird die Datei „<jada_dir>/MatrixCreator/bcsstk13_crs.out“ geöffnet.

8.3 Aufruf des Programmes

Da es sich bei dem Programm um ein paralleles Programm handelt, muss dieses auch mit einem entsprechenden Loader für parallele Programme gestartet werden. Auf dem Jump steht dazu das Programm „llrun“ zur Verfügung. Damit kann das Programm wie folgt gestartet werden:

llrun -p<#Prozessoren> jada14_1.exe [<Matrix-Datei>]

Der hier angegebene Parameter Matrix-Datei ist optional, er wird nur dann benötigt, wenn der Matrix-Index in der Parameterdatei auf „0“ gesetzt ist und man den Dateinamen der Matrix-Datei als Argument übergeben möchte.

im Folgenden Beispiel wird das Programm mit 16 Prozessoren gestartet. Als Eingabedatei wird die in der Parameterdatei spezifizierte Datei verwendet.
Beispiel: llrun -p16 jada14_1.exe

8.4 Konvertierungsprogramme

Im Rahmen der Diplomarbeit wurden einige Konvertierungsprogramme entwickelt, um Matrix-Daten in die vom Programm unterstützten Datei-Formate zu konvertieren.

Die Programme *CRSMatCreate* und *BandMatCreate* erzeugen aus einer Text-Datei Binär-Dateien mit den Matrizen im jeweiligen Format, das vom Programm erwartet wird. Die Textdatei sollte dabei folgende Form besitzen:

Zeile	Inhalt
1	Titel der Matrix
2	Dimension der Matrix
3	Koeffizienten der 1. Zeile der Matrix (durch Blank getrennt)
4	Koeffizienten der 2. Zeile der Matrix (durch Blank getrennt)
...	...

Tabelle 8.4: Aufbau der Eingabedateien für die Konvertierungsprogramme

Das Programm fordert nach dem Aufruf zur Eingabe des Namens der Eingabedatei auf. Der Ausgabedateiname entspricht dem der Eingabedatei mit der Endung „.out“.
Beispiel: test.inp -> test.out

Desweiteren wurde das Programm *mtx2out* entwickelt, das Matrizen im mtx-Format (Format des Matrix-Markets [10]) liest und CRS- und Band-Binärdateien erzeugt. Der Benutzer wird nach dem Aufruf aufgefordert den Namen der Eingabe einzugeben. Anschließend kann er wählen, welche Formate erzeugt werden sollen (0 = alle, 1 = nur CRS, 2 = nur Band). Die Namen der Ausgabedateien werden anhand des Namens der Eingabedatei generiert, in dem die Endung weggelassen und durch „_crs.out“ beziehungsweise „_band.out“ ersetzt wird.
Beispiel: test.inp -> test_crs.out beziehungsweise test_band.out

Zu beachten ist, dass das Programm für symmetrische Matrizen, die als oberes Dreieck angegeben sind entwickelt wurde. Das Programm speichert die Matrix nicht mehr als Dreieck sondern als „vollständige“ Matrix im jeweiligen Format.

Durch das Programm *mtx2out* ist es möglich alle Dateien des „Matrix Markets“ [10] zu verwenden. Hier sind die symmetrischen Matrizen als oberes Dreieck gespeichert und das Programm arbeitet somit einwandfrei.

Kapitel 9

Test der Programms

In diesem Kapitel werden einige Testbeispiele genauer erläutert, mit denen die Performance und das Laufzeitverhalten einzelner Routinen untersucht werden. Als Tools zur Performance-Messung wurden „Call Graph Profiling (gprof)“ [9] und „Hardware Counter Analysis (HPM)“ [4] eingesetzt.

9.1 Skalierungseigenschaft

In diesem Kapitel soll die Skalierungseigenschaft des Programms genauer untersucht werden. Dazu wurde das Programm mit verschiedenen Anzahlen von Prozessoren mehrfach auf 2 Matrizen angewendet und jeweils die Laufzeit gemessen. Die beiden Matrizen stellen beide eine 3D-Diskretisierung des Laplace-Operators auf einem Rechteckgitter dar. Bei diesem Problem ist die exakte Lösung bekannt und somit lassen sich die berechneten Ergebnisse sehr gut verifizieren. Die Matrizen haben die Dimensionen 40.950 und 188.750. Bei beiden Matrizen wurden 20 Eigenwerte mit einem Unterraum der Dimension 100 bestimmt. Als Gleichungssystemlöser wurde zunächst ein Diagonallöser und bei entsprechender Genauigkeit ein CG-artiger Löser verwendet.

Bei beiden Matrizen ergaben sich übereinstimmende Ergebnisse. Verdoppelt man die Anzahl der Prozessoren, so erreicht man zunächst einen sehr guten Speed-Up und die Laufzeit wird fast halbiert. Setzt man zu viele Prozessoren für das gewählte Problem ein, so erreicht man nahezu keinen Speed-Up mehr. Teilweise wird das Programm sogar langsamer. Dies ist dadurch zu erklären, dass das Problem bei zu vielen Prozessoren in zu kleine Stücke zerteilt wird, so dass der Kommunikationsaufwand im Verhältnis zum Berechnungsaufwand viel zu hoch wird.

Es lässt sich in beiden Statistiken weiterhin erkennen, dass man bei symmetrischen Matrizen einen kleinen Performancegewinn erzielen kann, wenn diese nur als unteres Dreieck verarbeitet werden. Um dies vergleichen zu können wurde jeweils der symmetrische und der unsymmetrische Algorithmus auf die symmetrische Systemmatrix angewendet. Der Performancegewinn ist allerdings relativ gering, da durch diese Art der Speicherung jeweils Matrix-Vektor-Produkte zur Multiplikation der Matrizen mit Vektoren verwendet werden müssen. Bei einer Speicherung der vollen Matrix kann man Matrix-Matrix-Produkte verwenden und somit effizientere Berechnungen durchführen.

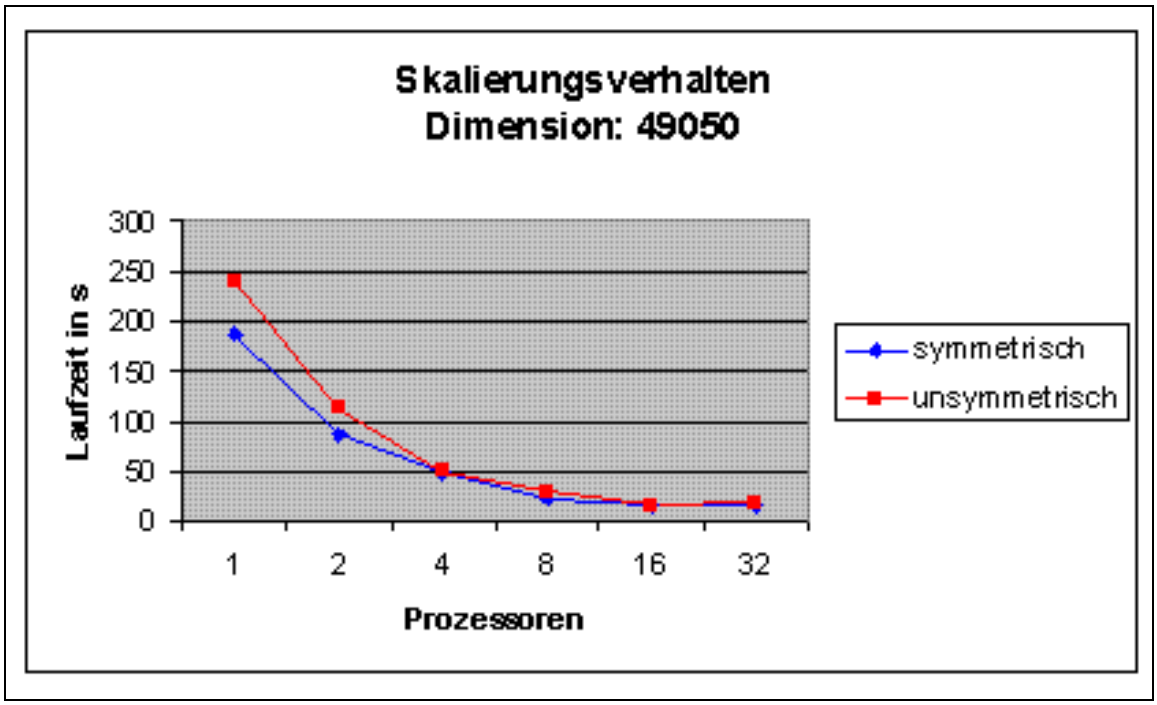


Abbildung 9.1: Skalierungsverhalten für Dimension 49.050

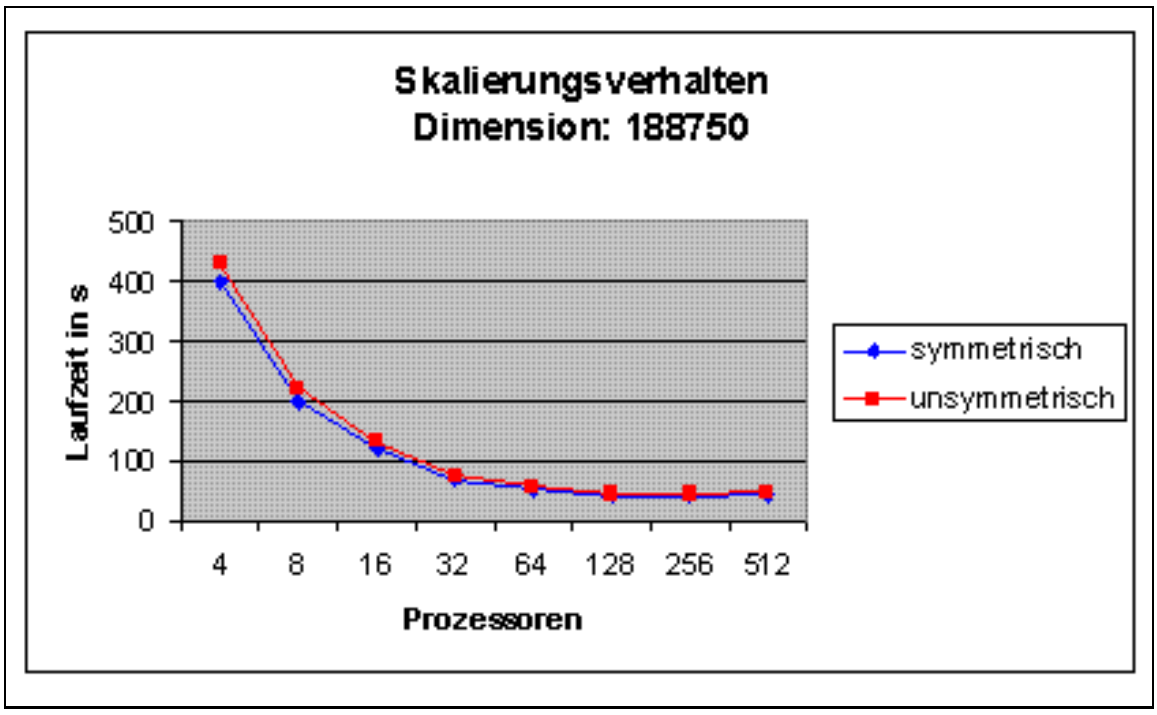


Abbildung 9.2: Skalierungsverhalten für Dimension 188.750

9.2 Performance

Der nächste Ansatzpunkt der Prüfungen sind die Anzahl der Operationen, die die Prozessoren pro Sekunde durchführen. Dazu wurde das Programm auf eine Matrix der Dimension 1.505.000 angewendet. Diese beschreibt wieder eine 3D-Diskretisierung des Laplace-Operators, allerdings auf einem wesentlich größeren Gebiet. Es wurden 20 Eigenwerte mit einer maximalen Unterraum-Dimension von 100 berechnet. Als Löser wurde zunächst ein Diagonallöser und anschließend das QMR-Verfahren ohne und später mit ILDLT-Vorkonditionierung verwendet. Zur Berechung der Eigenwerte wurde das Programm auf 256 Prozessoren ausgeführt.

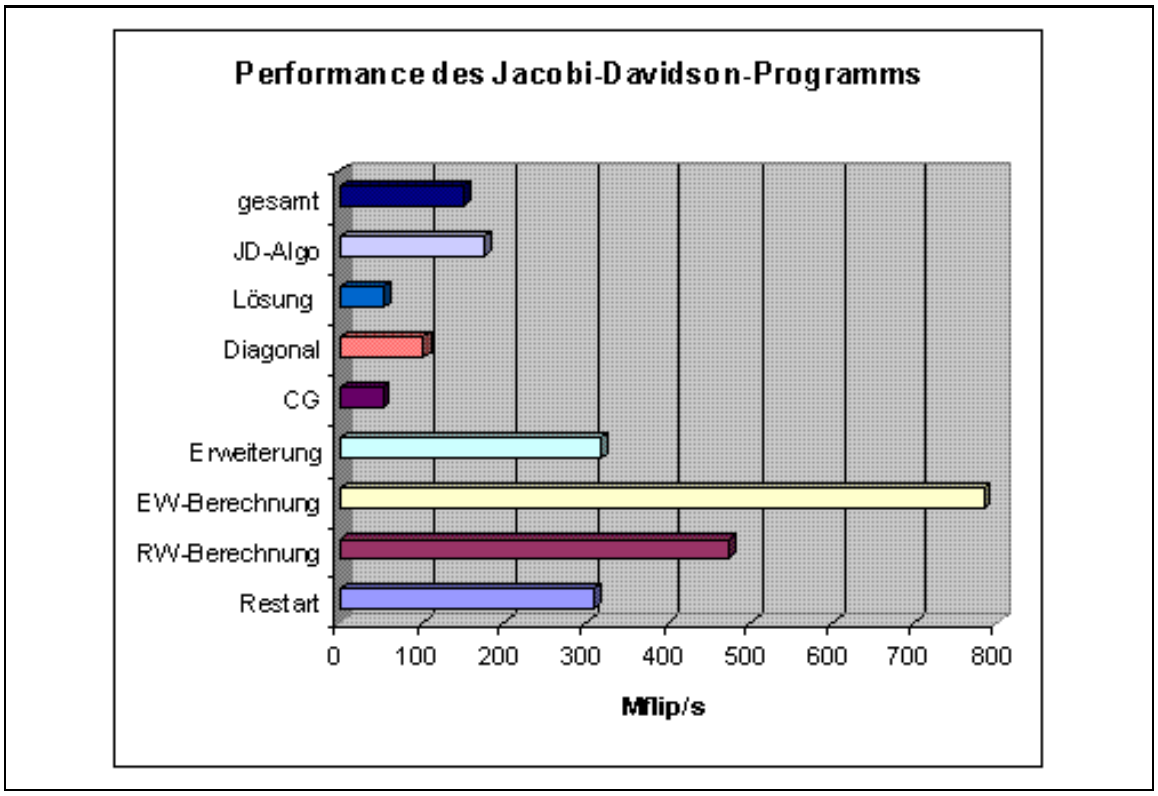


Abbildung 9.3: Performance Messungen für Dimension 1.505.000

Auf den ersten Blick lässt sich direkt erkennen, dass die größte Performance mit fast 800 Mflip/s bei der Berechnung der Eigenwerte erreicht wird. Dies ist offensichtlich, da die Berechnung der Eigenwerte mit Hilfe einer für den JUMP optimierten LAPACK-Routine berechnet werden. Weiterhin wird die Berechnung seriell auf jedem Prozessor durchgeführt und somit muss keinerlei Kommunikation statt finden.

Wie im nächsten Kapitel zu sehen ist, nimmt die Berechnung der Eigenwerte leider nur einen sehr kleinen Teil der Laufzeit in Anspruch. Viel wichtiger sind die Erweiterung des Unterraums und die Berechnung der Ritzwerte. Auch für diese beiden Aufgaben wird durch die Verwendung optimierter LAPACK Routinen eine gute Performance erreicht. Da bei beiden Aufgaben Kommunikation statt findet, werden hier nicht ganz so hohe Flip-Raten erreicht, wie bei der Eigenwertberechnung. Bei der Berechung der Ritzwerte werden ca. 475 Millionen Gleitkomma-Operationen pro Sekunde durchgeführt, bei der Unterraumerweiterung werden nur ca. 300 Mflip/s erreicht. Dies lässt sich dadurch erklären, dass im Programm eine Orthonormalbasis verwendet wird. Daher müssen die Korrekturvektoren vor der Unterraumerweiterung gegen den bestehenden Unterraum orthonormalisiert werden, was aber auf einem Parallelrechner sehr viel Kommunikation und Aufwand in Anspruch nimmt.

Eine weitere sehr wichtige Operation ist die Lösung der Korrekturgleichung. Hier wurden leider nur ca. 100 Mflip/s (Diagonallöser) beziehungsweise ca. 50 Mflip/s (QMR-Verfahren) erreicht. Dies

lässt sich dadurch erklären, dass hier selbst-entwicklete Routinen verwendet wurden, die nicht speziell für den JUMP optimiert sind.

9.3 Aufteilung der Laufzeit

Neben den reinen Performancedaten des Programms als Ganzes ist weiterhin interessant, wie sich die Laufzeit auf die einzelnen Teilaufgaben aufteilt. Dazu wurden bei der Verarbeitung der Laplace-3D-Diskretisierung, mit Dimension 1.505.000, Zeitmessungen der einzelnen Routinen durchgeführt. Es wurden zwei verschiedene Fälle untersucht, bei denen jeweils 20 Eigenwerte berechnet wurden.

Im ersten Test wurde dazu nur ein Diagonallöser eingesetzt. Wie zu erwarten ist, wurden sehr viele Iterationen benötigt. Die Iterationen konnten aber im Durchschnitt mit 0,2 Sekunden sehr schnell abgearbeitet werden. Der größte Rechenaufwand lag mit 44% bei der Erweiterung des Unterraums und der Berechnung der Ritzwerte mit 39%.

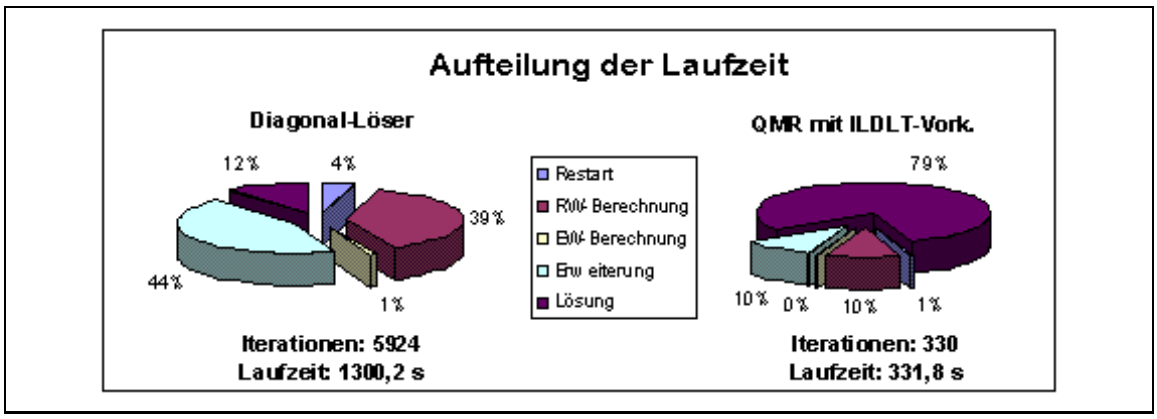


Abbildung 9.4: Laufzeitverhalten bei einfachem und aufwendigem Löser

Beim zweiten Test wurde als Löser das QMR-Verfahren mit ILUT-Vorkonditionierung verwendet. Die Anzahl der Iterationen konnte dadurch drastisch auf $1/20$ der vorherigen Anzahl reduziert werden. Die Laufzeit des Programms betrug nur noch ca. $1/4$ der vorherigen Laufzeit. Angestiegen ist die Laufzeit einer einzelnen Iteration, diese dauerte im Mittel mit ca. eine Sekunde fast fünf Mal so lange wie beim Diagonallöser. Dies spiegelt sich auch deutlich in der Aufteilung der Laufzeit wieder, denn die Rechenzeit für die Lösung der Korrekturgleichung betrug fast 80% der Gesamtlaufzeit.

9.4 Vergleich der Löser

Zur Lösung der Korrekturgleichung wurde für Bandmatrizen als zusätzlicher einfacher Löser der Bandlöser implementiert. Die folgenden Statistiken zeigen Messungen bei Matrizen der Dimension 5000 mit 50 Sub- und Superdiagonalen. Die Matrizen sind mit Zufallswerten besetzt, die von der Hauptdiagonale aus im Mittel abnehmen. Die Grafiken zeigen die Ergebnisse für Matrizen, bei denen die Koeffizienten im Mittel im Verhältnis zur Diagonale linear, quadratisch, kubisch, 4. Ordnung und exponentiell abnehmen. Es wird jeweils die Gesamtlaufzeit des Programms und die Anzahl der benötigten äußeren Iterationen angegeben.

Es lässt sich erkennen, dass außer bei der exponentiellen Abnahme jeweils die Laufzeit zunächst zunimmt. Dies ist dadurch zu erklären, dass ein Bandlöser wesentlich aufwändiger ist als ein Diagonallöser. Damit also die Laufzeit abnimmt, muss die Anzahl der Iterationen wesentlich reduziert werden.

Im Fall der linearen Abnahme kann die Anzahl der Iterationen nicht weit genug reduziert werden und pendelt sich bei ca. 9000 ein. Da die Löser immer aufwändiger werden, steigt somit die Laufzeit weiter an.

Bei der quadratischen Abnahme fällt die Anzahl der Iterationen nahezu linear ab. Da die Iterationskurve aber nur mit geringer Steigung fällt, reicht dies gerade aus um den zunehmenden Rechenaufwand der Löser auszugleichen. Dadurch erhält man ein fast konstantes Laufzeitverhalten.

Der Vorteil des Bandlösers lässt sich erst bei der Matrix mit kubischer Abnahme erkennen. Hier nimmt die Anzahl der Iterationen mit einer größeren Steigung linear ab. Dadurch nimmt auch die Laufzeit linear ab und bei einem Löser mit 23 Bändern erhält man ungefähr dieselbe Laufzeit wie beim Diagonallöser. Bei 25 Bändern kann die Laufzeit sogar noch weiter reduziert werden.

Betrachtet man die Ergebnisse der Untersuchung bei Abnahme vierter Ordung, so erkennt man einen deutlich größeren Vorteil des Bandlösers. Die Anzahl der Iterationen fällt sehr schnell linear ab, wodurch sich auch die Laufzeit relativ schnell linear verringert. Bereits mit einem Löser von 13 Bändern erreicht man eine kürzere Laufzeit als beim Diagonallöser. Bei Verwendung eines Bandlösers mit 25 Bändern kann man die Laufzeit fast um die Hälfte reduzieren.

Ein vollkommen anderes Verhalten als in den bisherigen Statistiken zeigt die Grafik für den exponentiellen Abfall. Hier sinkt die Anzahl der Iterationen direkt auf unter ein Drittel des Diagonallösers so drastisch ab, dass sich natürlich auch von vorne herein die Laufzeit verringert. Aber auch hier lässt sich der erhöhte Aufwand des Lösers erkennen, denn wie in den vorherigen Graphen auch schneiden sich die Kurve der Laufzeit und der Iterationen. Die Graphik zeigt weiterhin, dass es anscheinend bei sieben Bändern eine Art Minimalpunkt der Laufzeit gibt. Die Anzahl der Iterationen nimmt in den darauf folgenden Schritten nur noch so unwesentlich ab, dass die Laufzeit durch den erhöhten Aufwand wieder zunimmt. Im Vergleich zum Diagonallöser benötigt das Verfahren beim Bandlöser mit sieben Bändern nur noch weniger als ein Sechstel der Laufzeit.

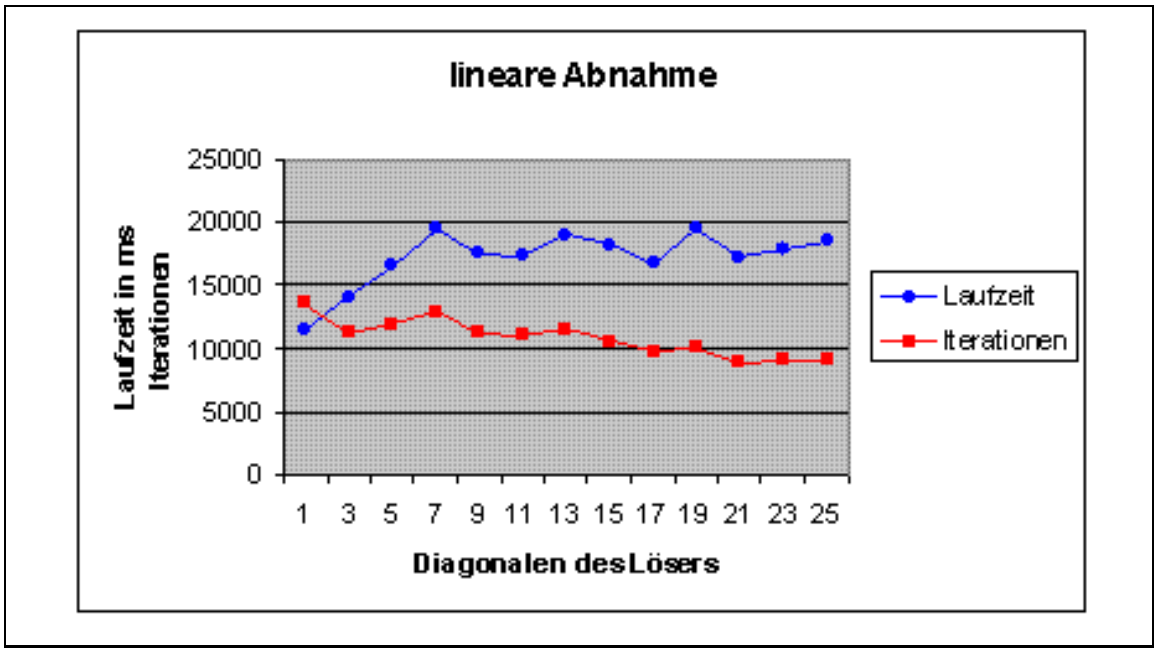


Abbildung 9.5: Test des Bandlösers bei linearer Abnahme der Koeffizienten

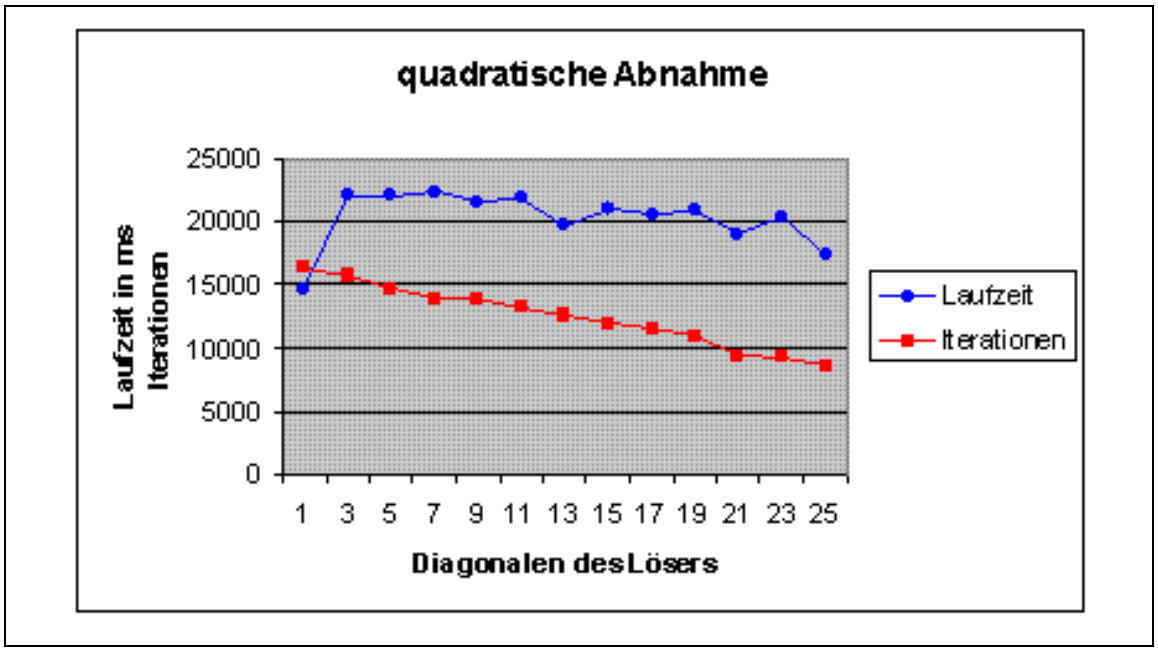


Abbildung 9.6: Test des Bandlösers bei quadratischer Abnahme der Koeffizienten

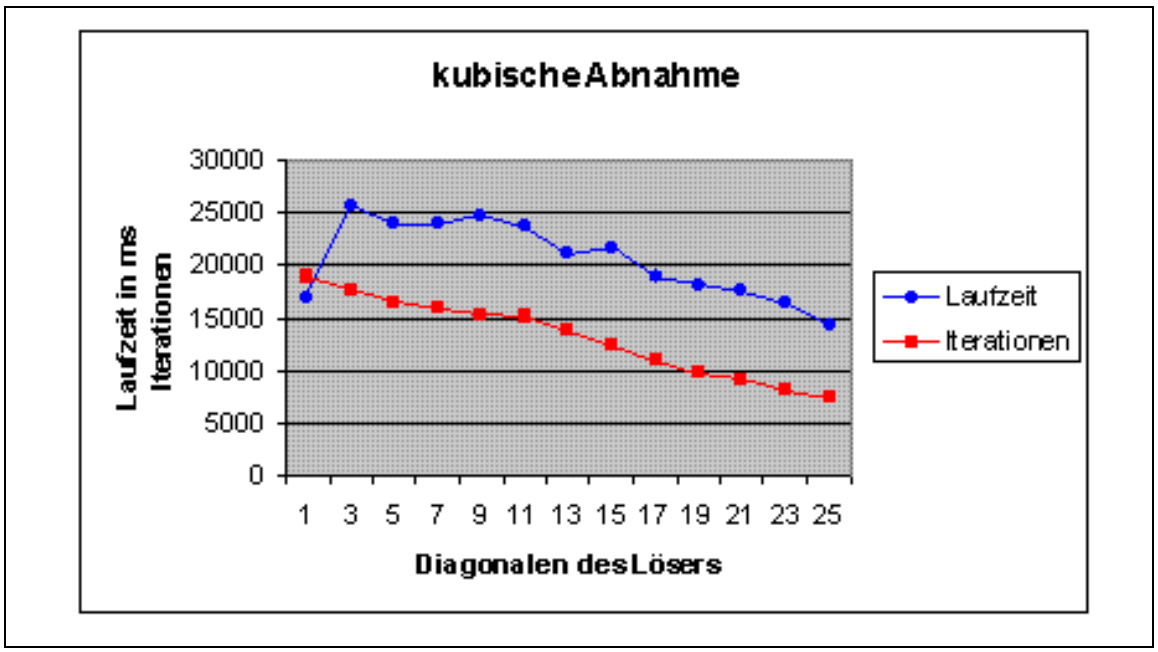


Abbildung 9.7: Test des Bandlösers bei kubischer Abnahme der Koeffizienten

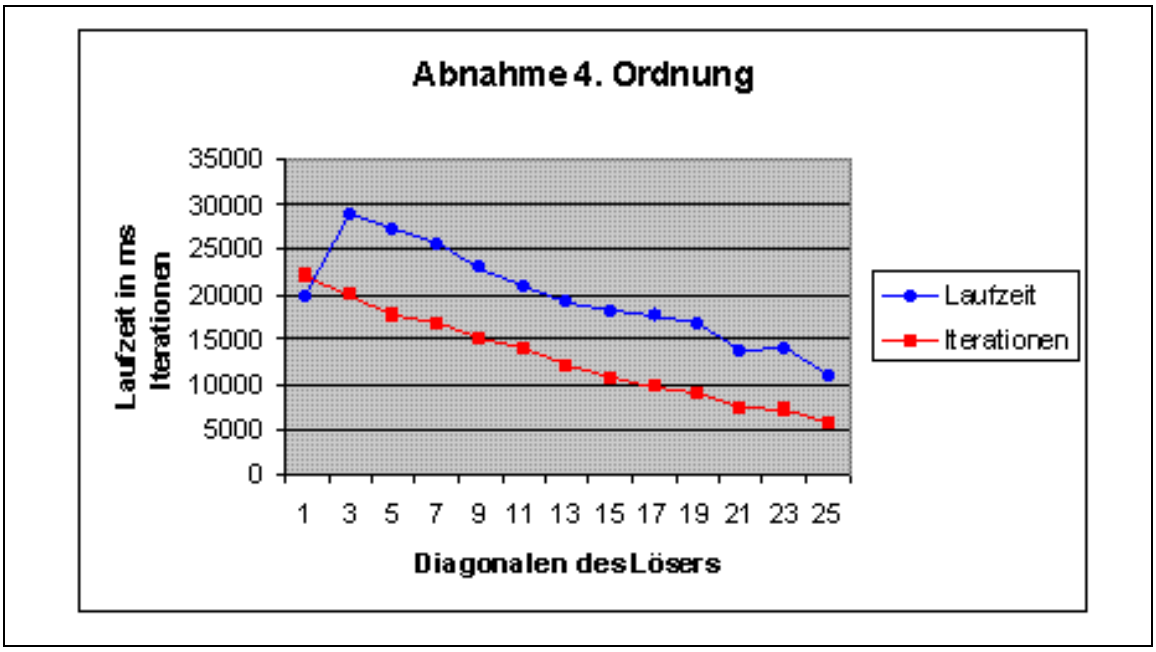


Abbildung 9.8: Test des Bandlösers bei Abnahme 4. Ordnung der Koeffizienten

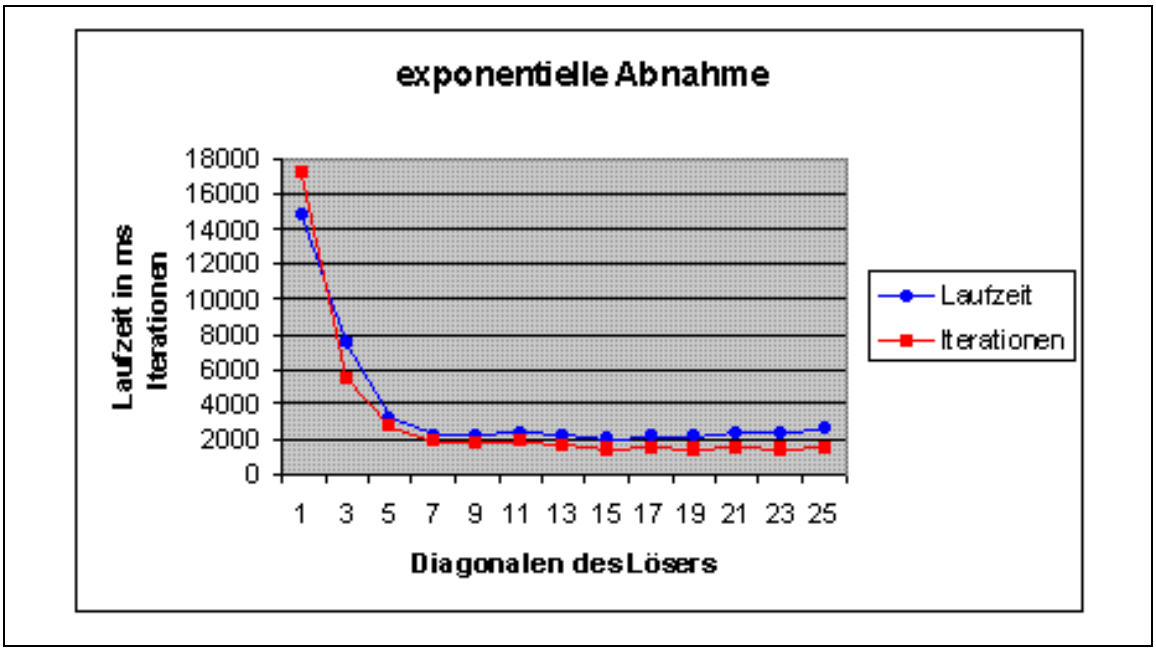


Abbildung 9.9: Test des Bandlösers bei exponentieller Abnahme der Koeffizienten

9.5 Vergleich der Matrix-Formate

Im Kapitel 5.10 wurde bereits theoretisch diskutiert, wann sich der Einsatz des Band-Formates lohnt. In der folgenden Statistik wurden Zufallsmatrizen der Dimension 5000, mit der bereits beschriebenen Besetzung, generiert. Die Matrizen haben jeweils 50 Sub- und Superdiagonalen, allerdings eine verschiedene Besetzungswahrscheinlichkeit innerhalb dieser Bänder. Die erste Matrix ist im inneren Bereich voll besetzt, die zweite nur noch zu 90%, die letzte nur noch zu 10%. Es wurde für beide Formate jeweils der Speicherbedarf der Matrix auf der Festplatte, so wie das Laufzeitverhalten der Matrix-Vektor-Produkt-Routine gemessen.

In der Grafik lässt sich erkennen, dass sowohl der Speicheraufwand als auch das Laufzeitverhalten für ein Matrix-Vektor-Produkt beim Bandmatrix-Format für alle Besetzungsgrade konstant ist. Dies

ist offensichtlich, denn beim Bandmatrix-Format werden unabhängig von der Besetzung immer alle Koeffizienten der inneren Bänder gespeichert. Weiterhin wird bei den Produkten auch mit all diesen Koeffizienten gerechnet.

Das CRS-Format hingegen speichert nur die Nicht-Null-Koeffizienten ab. Daher nimmt der Speicherbedarf mit linear abfallender Besetzung auch linear ab. Weiterhin nimmt das Laufzeitverhalten der Matrix-Vektor-Produkt-Routine linear ab, da diese bei der Multiplikation auch nur die Nicht-Null-Koeffizienten berücksichtigt und somit immer weniger Multiplikationen durchführen muss.

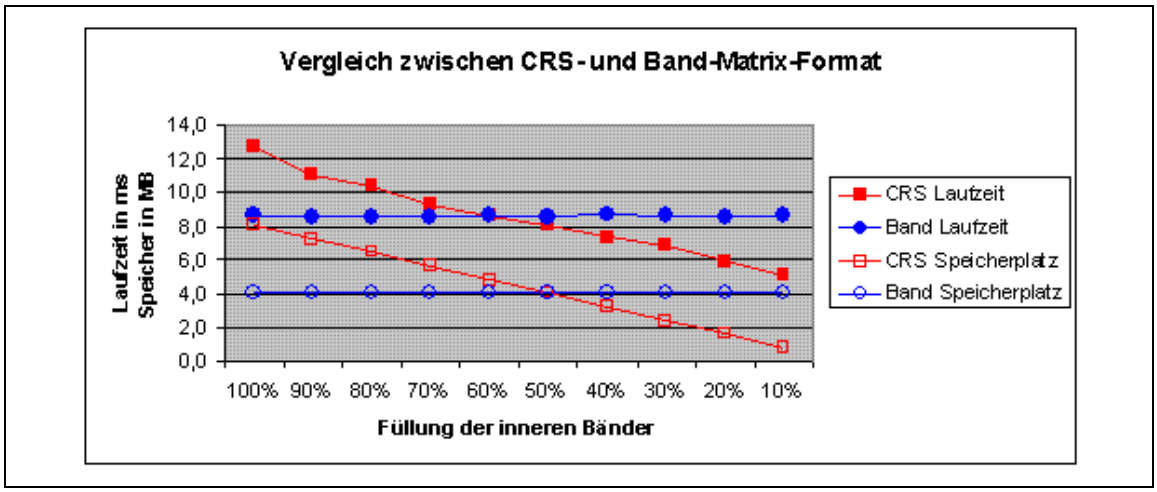


Abbildung 9.10: Vergleich zwischen CRS- und Band-Matrix-Format

Das CRS-Format eignet sich bei voller Besetzung der inneren Bänder dennoch schlechter, da für jeden Koeffizienten zusätzlich der Spaltenindex abgespeichert werden muss. Weiterhin muss beim Matrix-Vektor-Produkt jeweils, anhand des Spaltenindex, zweifach indiziert auf die rechte Seite zugegriffen werden. Eigentlich würde man hier gegenüber dem Band-Matrix-Format eine wesentlich schlechtere Performance erwarten. Anscheinend kann die Doppel-Indizierung aber hervorragend vom Compiler optimiert werden.

Die Statistik bestätigt die in Kapitel 5.10 theoretisch hergeleiteten Aussagen und zeigt, dass für Matrizen, die im inneren Band-Bereich einen sehr hohen Besetzungsgrad haben, das Band-Matrix-Format besser geeignet ist. Bei Matrizen, die in diesem Bereich sehr dünn besetzt sind, eignet sich wie erwartet das CRS-Format besser. Es lässt sich erkennen, dass die Grenze, bei der das Format gewechselt werden sollte, im Bereich eines Besetzungsgrades von ca. 50% - 60% liegt.

9.6 Instabilität der QMR-Verfahren

Beim Tests des Programms für innere Eigenwerte wurde deutlich, dass das Programm viel zu viele Iterationen benötigt, um diese zu berechnen. Eine genauere Untersuchung des Problems ergab, dass die Korrekturgleichung nicht genau genug gelöst wird. Daraufhin wurde das TFQMR-Verfahren überprüft. Es wurde festgestellt, dass das TFQMR-Verfahren richtig implementiert ist und alle Iterationsschritte korrekt berechnet werden. Allerdings konvergiert das Verfahren gegen einen falschen Wert. Dies lässt sich dadurch erklären, dass das gestellte Problem stark singulär ist und die QMR-Verfahren für stark indefinite nicht geeignet sind.

Bei der Berechnung der Eigenwerte der unsymmetrischen 6×6-Matrix:

$$A = \begin{Bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 2 & 1 & 0 & 0 & 0 \\ 0 & 2 & 3 & 1 & 0 & 0 \\ 0 & 0 & 2 & 4 & 1 & 0 \\ 0 & 0 & 0 & 2 & 5 & 1 \\ 0 & 0 & 0 & 0 & 2 & 6 \end{Bmatrix}$$

muss die Korrekturgleichung

$$(I - u_k u_k^T)(A - \tau I)t = -r_k$$

mit dem Ziel $\tau = 3.5$ und der Eigenvektor-Näherung

$$u_k = \begin{pmatrix} -0.3947196 & 0.6080864 & -0.5193272 & 0.4305681 & -0.1260047 & 0.0588260 \end{pmatrix}^T$$

gelöst werden. Als Residuum ergibt sich mit Hilfe des Rayleigh-Quotienten:

$$\begin{aligned} r_k &= Au_k - < u_k, Au_k > u_k \\ &= \begin{pmatrix} 0.2223960 & -0.1065037 & 0.1006388 & 0.5477639 & 0.2928209 & .0996008 \end{pmatrix} \end{aligned}$$

Das TFQMR-Verfahren liefert nach sechs Iterationen das folgende Ergebnis:

$$t_{qmr} = \begin{pmatrix} -0.0727512 & 0.0874120 & 0.0978740 & 0.0364495 & 0.2826383 & -0.1890658 \end{pmatrix}^T$$

In der fünften und sechsten Iteration wurden die ersten sechs Nachkommastellen der Lösung nicht mehr geändert, so dass man davon ausgehen kann, dass das Verfahren auch wirklich gegen t_{qmr} konvergiert.
Die korrekte Lösung der Korrekturgleichung ist aber:

$$t_{kor} = \begin{pmatrix} -0.1223039 & 0.0338113 & 0.0772035 & -0.2236970 & 0.0465524 & -0.0354436 \end{pmatrix}^T$$

Die Vektoren schließen einen Winkel von

$$\angle(t_{kor}, t_{qmr}) = 72,35^\circ$$

ein. Sie sind also vollkommen unabhängig voneinander.

Die QMR-Verfahren können also nicht zur Berechnung innerer Eigenwerte verwendet werden. Verwendet man den Bandlöser zur Lösung der Korrekturgleichung, so konvergiert das Verfahren innerhalb von fünf Iterationen und liefert die korrekten Eigenwerte.

9.7 Vergleich der Konvergenz

Schließlich wurde noch ein Test für alle vier Varianten: Berechnung ...

- äußere Eigenwerte einer symmetrischen Matrix
- innere Eigenwerte einer symmetrischen Matrix
- äußere Eigenwerte einer unsymmetrischen Matrix
- innere Eigenwerte einer unsymmetrischen Matrix

durchgeführt. Dazu wurde die folgende Matrix der Dimension 250.000 verwendet:

$$A_{symm} = \begin{pmatrix} 1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & 249999 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & 250000 \end{pmatrix}$$

Für den unsymmetrischen Test wurde die Matrix A_{uns} verwendet, die bis auf die Superdiagonale der Matrix A entspricht. Die Superdiagonale ist bei A_{uns} anstatt mit Einsen mit Zweien besetzt.

Es wurden jeweils 20 Eigenwerte bei einer maximalen Unterraum-Dimension von 40 bestimmt. Als Gleichungssystemlöser wurde der Diagonallöser verwendet. Da die Matrix in Hessenberg-Form ist könnte man sie auch sehr effizient mit den klassischen Eigenwertverfahren lösen. Sie eignet sich trotzdem sehr gut zur Untersuchung des Konvergenzverhaltens des Jacobi-Davidson-Verfahren, da dieses auch bei anderen Besetzungsstrukturen ähnlich aussieht.

Die gesuchten Eigenwerte wurden immer mit wenigen Iterationen berechnet. Die Berechnung der Eigenwerte der unsymmetrischen Matrix benötigte aber deutlich mehr Iterationen, als die der symmetrischen Matrix. Dies lässt sich dadurch erklären, dass das Jacobi-Davidson-Verfahren für symmetrische Matrizen im Allgemeinen deutlich schneller konvergiert als für unsymmetrische Matrizen. Weiterhin eignet sich der Diagonallöser schlechter für die unsymmetrische Matrix, als für die symmetrische, so dass die Korrekturgleichung ungenauer gelöst wird.

Die Berechnung der inneren Eigenwerte der unsymmetrischen Matrix benötigte noch einmal mehr Iterationen, auch hier liegt der Grund wohl im Konvergenzverhalten des Verfahrens, das für innere Eigenwerte etwas schlechter als für äußere ist.

Es lässt sich erkennen, dass die Laufzeiten pro Iteration bei inneren Eigenwerten ein wenig höher als bei äußeren sind. Dies ist dadurch zu erklären, dass die Erweiterung der Unterräume und die Berechnung der Ritzvektoren bei harmonischen Ritzwerten deutlich aufwändiger ist als bei normalen Ritzwerten.

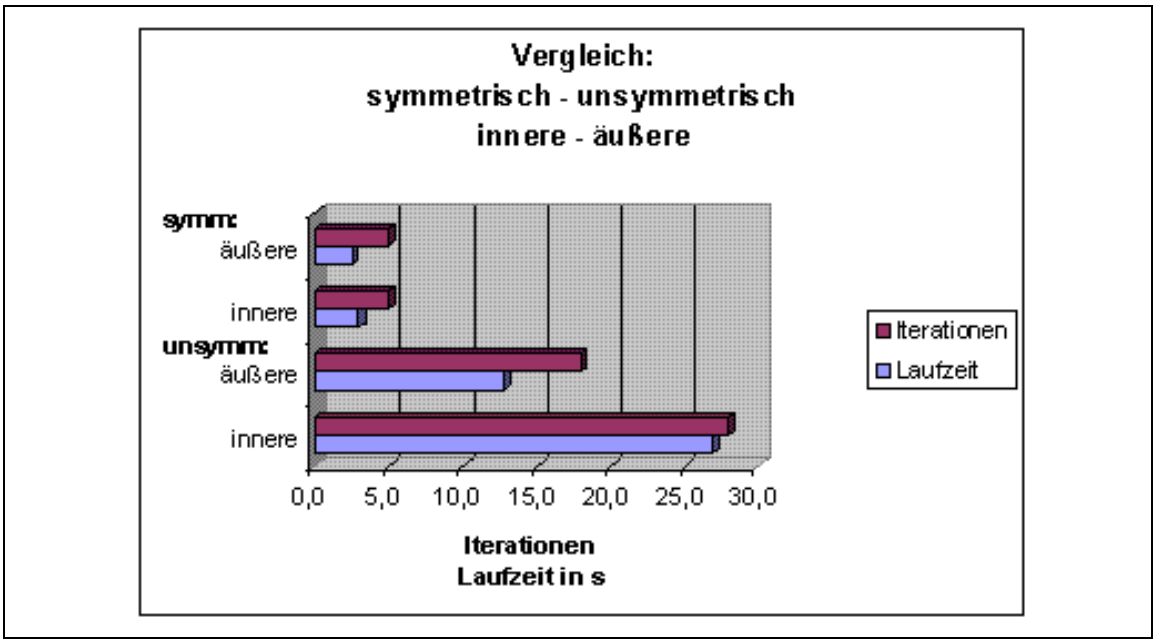


Abbildung 9.11: Vergleich der Konvergenz

Die nächsten Abbildungen zeigen der Verlauf der Eigenwertnäherungen, bei den vier Untersuchungen. Näherungen, die zu weit vom Konvergenzziel entfernt sind, wurden aus Gründen der Übersichtlichkeit nicht in die Grafik eingezeichnet.

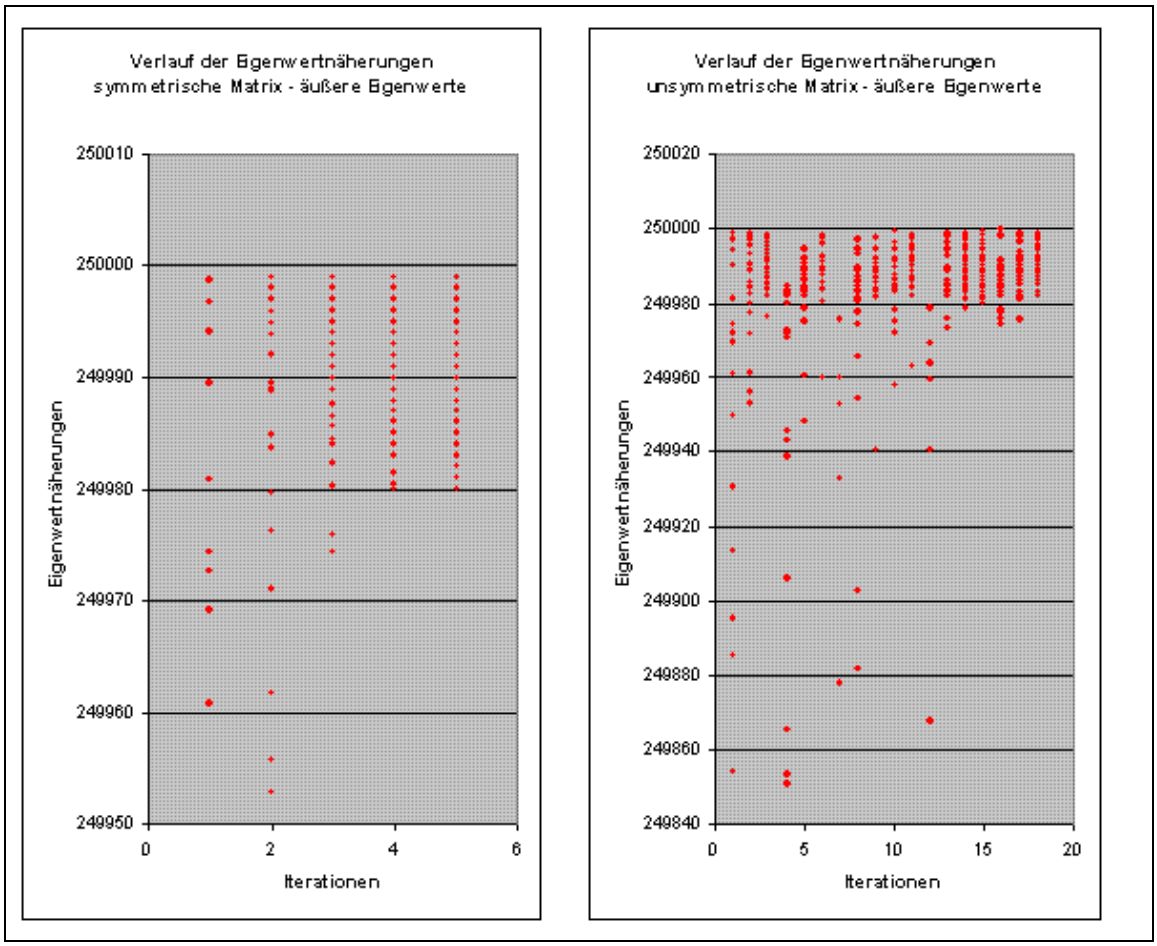


Abbildung 9.12: Verlauf der äußeren Eigenwertnäherungen

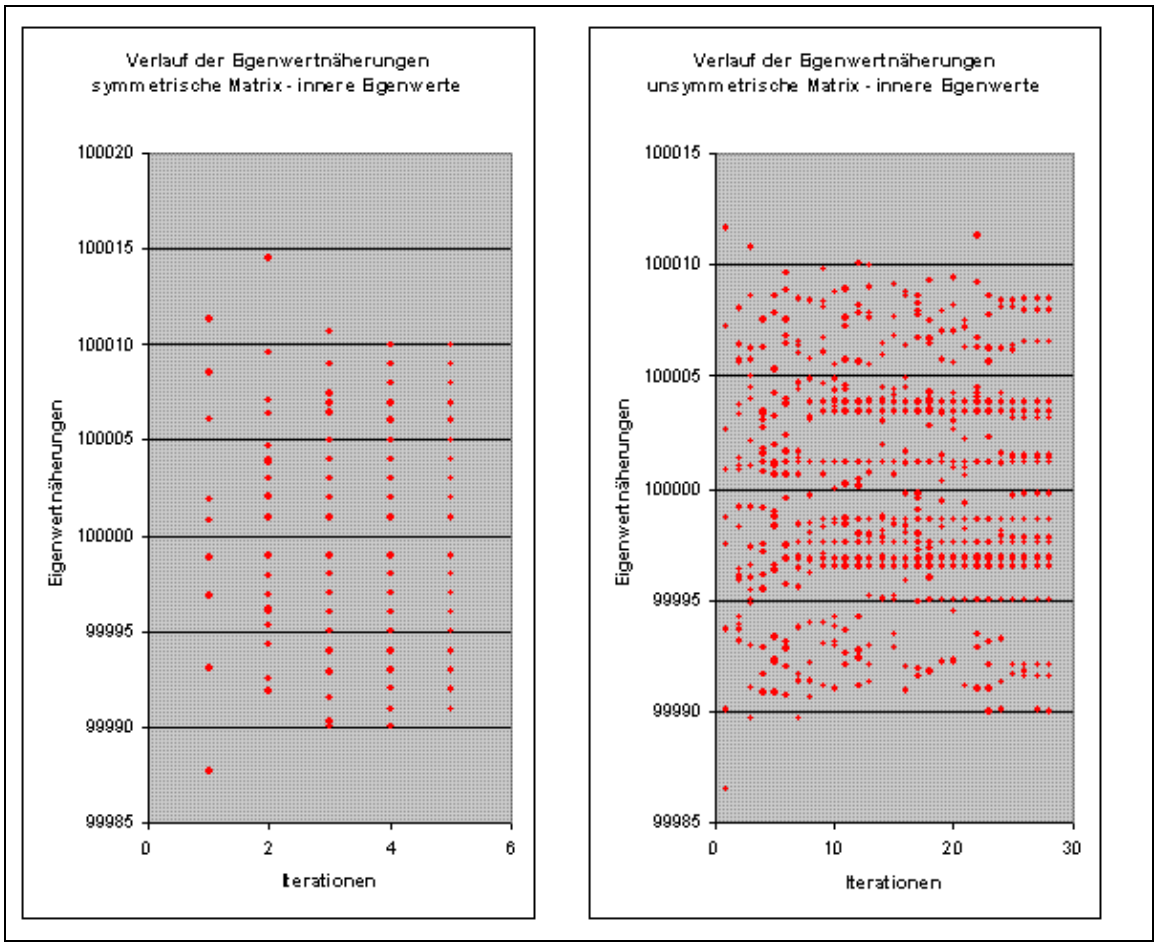


Abbildung 9.13: Verlauf der inneren Eigenwertnäherungen

Kapitel 10

Zusammenfassung und Ausblick

In der Diplomarbeit wurde das Programm so strukturiert, dass die Matrix-Verwaltung vollkommen vom Hauptalgorithmus abgekoppelt ist. Dadurch ist es nun sehr leicht möglich neue Matrix-Formate in das Programm aufzunehmen. Ein Format zur effizienten Verarbeitung von Bandmatrizen wurde bereits implementiert.

Auch die Lösung der Korrekturgleichung wurde vom Hauptalgorithmus abgekoppelt, so dass in Zukunft auch sehr leicht neue Löser eingebunden werden können. Für die Bandmatrizen wurde bereits ein neuer Löser, ein Bandlöser implementiert.

Das Programm wurde außerdem um die Möglichkeit erweitert innere Eigenwerte zu bestimmen. Hier gab es teilweise Probleme bei der Lösung der Korrekturgleichung, da die QMR-Verfahren nicht stabil waren. Es wäre also sinnvoll das Programm um einen stabileren Löser, beispielsweise BiCGStab [12] zu erweitern.

Zusätzlich zu den bisherigen Möglichkeiten, wurde die Verarbeitung von unsymmetrischen, symmetrisierbaren Matrizen implementiert. Diese Implementierung bildet bereits die Grundlage um das Programm so zu erweitern, dass die Eigenwerte allgemeiner unsymmetrischer Matrizen bestimmt werden können. Dazu müssen im Wesentlichen nur die Eigenwerte komplex gespeichert werden. Es ist aber bei geschickter Implementierung kein zusätzlicher Speicherplatz notwendig, da bekannt ist, dass zu jedem Eigenwert auch ein konjugiert komplexer Eigenwert existiert. Die reellen Speicherplätze der beiden Eigenwerte können also geteilt werden, um den Real- und Imaginärteil der beiden Eigenwerte abzuspeichern.

Literaturverzeichnis

[1] A. Basermann, *Iterative Verfahren für dünnbesetzte Matrizen zur Lösung technischer Probleme auf massiv-parallelen Systemen*, Jül-3015, Forschungszentrum Jülich GmbH (1995)

[2] M. Bücker, *A transpose-free 1-norm quasi-minimal residual algorithm for non-Hermitian linear system*, IB-9706, Forschungszentrum Jülich GmbH (1997)

[3] E. R. Davidson, *The iterative calculation of a few, of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices*, Journal of Computational Physics (1975)

[4] L. DeRose, *Hardware Performance Monitor (HPM) Toolkit*, Advanced Computing Technology Center IBM Research (2003)

[5] G. Dikta, *Stochastik*, Fachhochschule Aachen, Vorlesungsskript

[6] W. Erkens, *Programmieren in Fortran 90/95*, Forschungszentrum Jülich GmbH (1999)

[7] A. Goeke, *Parallele Teilraumverfahren zur Bestimmung von Eigenwerten im Inneren des Spektrums*, Forschungszentrum Jülich GmbH (2000)

[8] C. G. J. Jacobi, *Über ein leichtes Verfahren, die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch auflösen*, Journal für die reine und angewandte Mathematik (1846)

[9] B. Mohr, *Perfomance Analysis Tools - Call Graph Profiling*, http://jumpdoc.fz-juelich.de (2003)

[10] National Institute of Standards and Technology *Matrix Market*, http:/math.nist.gov/MatrixMarket

[11] S. Pawelke, *Lineare Algebra*, Fachhochschule Aachen, Vorlesungsskript

[12] M. Reißel, *3D Eddy-Current Computation Using Krylov Subspace Methods*, Universität Kaiserslautern

[13] M. Reißel, *Spezielle Methoden der angewandten Mathematik*, Fachhochschule Aachen, Vorlesungsskript

[14] University Tennessee, *MPI: A Message-Passing Interface Standard*, University Tennessee (1995)

[15] University Tennessee, *LAPACK Users' Guide, Third Edition*, University Tennessee (1994)

[16] University Tennessee, *ScaLAPACK Users' Guide*, University Tennessee (1997)

[17] H. A. van der Vorst, *Computational Methods for large eigenvalue problems*, Handbook of Numerical Analysis, Vol. VIII (2001)